Persistent State Service 2.0

Joint Submission

Expersoft Corporation
IONA Technologies
Objectivity, Inc.
Oracle Corp.
Persistence Software, Inc.
POET Software Corp.
Sun Microsystems
TIBCO Software
Unidata, Inc.
Visigenic Software

with collaboration and support from

IBM Corp.
Novell, Inc.
Versant Object Technology
Windward Solutions, Inc.

OMG TC Document ORBOS/97-11-05
10 November 1997

TABLE OF CONTENTS

1. PREFACE

1.1. Submitting Companies

The following companies are pleased to jointly submit this specification in response to OMG Persistent State Service 2.0 RFP (doc orbos/97-06-07):

- Expersoft
- IONA
- Objectivity, Inc.
- Oracle Corp.
- Persistence Software, Inc.
- POET Software Corp.
- Sun Microsystems, Inc.
- TIBCO Software, Inc.
- Unidata, Inc.
- Visigenic Software Inc.

In addition, the following companies have collaborated in development of the technology and jointly support this specification:

- IBM Corp.
- Novell, Inc.
- Versant Object Technology
- Windward, Inc.

1.2. Guide to the Submission

Sections 1 through 3 of this document provide background for this OMG submission. Section 4 introduces our approach to persistence and how it relates to the old POS, the POA, and to database standards. Section 5 is an overview of the PSS we propose from a programmer's point of view. Section 6 provides more detail and explains how the PSS would be implemented. Section 7 defines IDL for the CosPersistent module. Section 8 references related documents. Sections 9 through 11 define ODMG APIs we propose for C++, Smalltalk, and Java.

1.3. Missing Items

See the the discussion in the specification and the list of open issues at the end of Section 6 for possible areas for further work on this proposal.

1.4. Conventions

Standard narrative appears using this font.

```
IDL and pseudo-code examples appear using this font.
```

1.5. Submission Contact Points

All questions about the joint submission should be directed to:

Contact: Joey Garon
Company: Expersoft Corporation
Address: 5825 Oberlin Drive, Suite 300, San Diego CA 92121
Phone: (619) 824-4135
Email: jgaron@expersoft.com

Contact: Irv Traiger
Company: IBM Corporation
Address: IBM Santa Teresa Laboratory, Mailstop D406 555 Bailey Avenue, San Jose CA 95141
Phone: (408) 463-4022
Email: traiger@us.ibm.com

Contact: Martin Chapman
Company: IONA Technologies PLC
Address: 8-10 Lower Pembroke Street, Dublin 2, Ireland
Phone: +353 1 662 5255
Email: mchapman@iona.com

Contact: Michael MacKay
Company: Novell
Address: 2180 Fortune Drive, San Jose CA 95131
Phone: (408) 577-6368
Email: mmackay@novell.com

Contact: Drew Wade
Company: Objectivity, Inc.
Address: 301B E. Evelyn Avenue, Mountain View CA 94041-1530
Phone: (650) 254-7113
Email: drew@objectivity.com

Contact: Jim Trezzo
Company: Oracle Corp.
Address: 500 Oracle Parkway, Redwood Shores CA 94065
Phone: (650) 506-8240
Email: jtrezzo@us.oracle.com

Contact: J Patrick Ravenel
Company: Persistence Software Inc.
Address: 1720 South Amphlett Blvd. Suite 300 San Mateo CA 94402
Phone: (619) 592-9220
Email: patrick@persistence.com

Contact: Olaf Schadow
Company: POET Software Corp.
Address: 999 Baker Way, Suite 100 San Mateo CA 94404
Phone: (650) 286-4640
Email: olafs@poet.com

Contact: Rick Cattell
Company: Sun Microsystems UCUP02-201
Address: 2550 Garcia Avenue Mountain View CA 94043
Phone: (408) 343-1409
Email: rcattell@eng.sun.com

Contact: Arvola Chan
Company: TIBCO Software Inc.
Address: 3165 Porter Drive, Palo Alto CA 94302
Phone: (650) 846-5046
Email: arvola@tibco.com

Contact: Francois Bancilhon
Company: Unidata
Address: 1099 18th street, Suite 2200, Denver CO, 80802
Phone: (303) 382 5447
Email: francois@unidata.com

Contact: Craig Russell
Company: Versant Object Technology
Address: 6539 Dumbarton Circle, Menlo Park CA
Phone: (510) 789-1662
Email: clr@versant.com

Contact: Jeff Michkinshky
Company: Visigenic Software Inc
Address: 951 Mariners Island Blvd San Mateo CA 94404
Phone: (650) 312-5158
Email: jeffm@visigenic.com

Contact: Jeff Eastman
Company: Windward Solutions Inc
Address: P.O. Box 819, Redwood City CA 94064
Phone: (415) 298-0023
Email: jeff@windwardsolutions.com

2. PROOF OF CONCEPT

The submitters have not yet completed a formal working demonstration of a PSS but the design is not simply abstract, but rather has evolved over the years, starting in 1991, under the auspices of ODMG. During this time, not only have there been design sessions with multiple vendors, end users, and integrators, but there have also been significant implementations. This last is very important. It's only when a design is implemented and delivered to a user that we first begin to understand how effective it is at actually meeting the user's needs. Several iterations on this design have occurred, over a period of 6 years, all based on the experience of vendors and users.

Proof of concept, as required by the RFP, will be provided at a place and time to be agreed with the ORBOS TF. This will include at least two implementations using at least an ODBMS (Objectivity, POET, etc.) and an RDBMS (Persistence, JavaSoft Java/Blend, with major RDBMSs).

See also *"Implementation Examples"*

3. RESPONSE TO RFP REQUIRMENTS

The following is a list of requirements from the Persistent State Service RFP (doc orbos/97-06-07), specifying how this submission is responsive to the RFP.

3.1. Scope of Proposals Sought

Interfaces for implementing Persistent States (PSs). A PS object is simply a CORBA object whose state is preserved by the PSS.

See section "COS Persistent Module".

A means, compatible with IDL, for defining persistent state and interfaces (sometimes called "schema")

This specification uses the proposed IDL extensions for specifying an object's state that are in the Objects-by-Value submission from: IBM, Netscape, Novell and Visigenic. We expect that Objects-by-Value will be adopted by OMG prior to the adoption of the PSS Service. We would then attempt to align this specification with the appropriate parts of Objects-by-Value. We are also looking at requirements for schema definition that may go beyond what we expect to be in Objects-by-Value. The current ODMG, Object Definition Language (ODL) allows you to specify relationships, which are not covered in the current Objects-by-Value submissions. We are discussing the possible inclusion of relationships in this submission.

Interfaces supporting one or more datastores. A datastore is a repository for persistent state. These interfaces must allow object implementations to achieve and maintain their persistence without regard to the implementation choice of the datastore.

This specification supports more than one datastores. Each ObjectManager is associated with a particular datastore(of a particular type). Examples of datastores are:

- *Flat File System*
- *Object Database Management System*
- *Relational Database Management System*
- *Object-Relational Database Management System*
- *Hierarchical Database Management System*

Interfaces supporting the management of persistent object lifecycle, including creation and deletion of objects with persistent state. The submission should be compatible with the CORBAServices Lifecyle specification, if at all possible: differences and additions should be fully explained.

See "Related Services" in Section 6.

Support for efficient access, via CORBA, to the states of a large number of possibly fine-grained objects.

See "Managing Many Objects Efficiently" in Section 5.

3.2. Internal Requirements

All capabilities are accessible through IDL specified interfaces.

3.3. External Requirements

This specification supports the Object Transaction Service. See Section 6.

3.4. Optional Requirements

3.4.1. Schema and Data Administration

We define schemas using IDL with Objects-by-Value extensions. See Section 6.

3.4.2. Additional Interfaces

We define the semantics of persistence objects in specific programming languages. See the appendices.

3.4.3. Additional Optional Services Integration

See “Related Services” in Section 6.

4. PERSISTENT STATE SERVICE

4.1. Introduction

This specification is in response to OMG PSS2.0 RFP (doc orbos/97-06-07), which calls for a CORBAservice to provide support for maintaining long-term storage of the states of CORBA objects and to replace the Persistent Object Service.

Persistence is an extremely important capability. In practice most commercial business users require some way to save the information within their object systems, and later access that information. We use the term "persistent" to mean a mechanism which preserves the value of an object's state (see below for definition of state) longer than a single process. Such persistent storage may be implemented directly by the application builder, but that can be a significant burden. It is desirable to provide a common service, not only to save the effort of individual implementations, but also to make the persistent storage facility available for general and common, shared use. In fact, sharing objects among multiple users is one of the key goals of Object Technology.

POS was adopted early in the OMG history. It has seen only one implementation, and is generally viewed, today, as not meeting the needs of object users. In particular, it has the following problems:

- *Direct Client Management of Persistence*

  In POS, the client (end user) specifically invokes operations to save and later restore the state of objects. Instead, experience has shown that clients would like to have the persistent storage occur automatically. Even implementers of CORBA objects desire as much automatic operation as possible.

- *Two Level Storage Model*

  The POS model directly exposes to users two levels of state information for objects. One level is the state currently in use in the executing object ("in memory state"), while the other is the state as stored in some storage facility ("on disk state"). The user (client) must keep track of these two, and manually manage the moving of information ("save" and "restore") to and from the disk and in-memory states. This manual management is difficult and error prone. Instead, experience of the last years has taught us that users (and even object implementers) would prefer a model in which there is but one level, the object with its state, while the persistent storage mechanism manages the mechanisms to move information back and forth, providing the desired and expected semantics without the user's worrying about manually caching and un-caching state information.

  It is noted that more complex applications with more demanding concurrency control and multi-user access, may require more direct control of the caching policy than is possible with automatic mechanisms. In these cases we may need to allow the object implementers the ability to synchronize object state in memory with the persistent state in the datastore, in between transactional boundaries. This is currently an open discussion area.

- *Complex Internal Architecture*

  The POS specification, in an attempt to provide very wide flexibility, developed an internal architecture with several components, all of which are required for a compliant implementation. Unfortunately, this complexity goes beyond what some wish to provide, and certainly beyond the fundamental need of maintaining persistent state. This raises a barrier to implementation, and to use of implementations.

- *Problems with Encapsulation*

  The above-described complex internal architecture required several different objects (persistent object service, data store, etc.) to communicate. These communications, among other things, required these objects (and even clients) to pass among themselves the internal state of another object (the persistent object). Exposing such an internal implementation violates encapsulation.

- *Poor Integration with Existing Persistent Storage Facilities*

  The design of POS, in trying to serve everyone and a broad set of capabilities, does not mesh well with existing persistent storage products, including those based on flat files, (object-) relational databases, and object databases. This is not only a hindrance to implementation, but makes the implementations awkward, more difficult to use, and possibly slower.

Therefore, the OMG ORBOS has chosen to issue an RFP for PSS2, to replace POS. It specifically calls for a simpler service, focusing on the basic needs of persistent storage, supportive of multiple technologies, and designed to serve the object implementer, as we discuss next.

4.2. Who is addressed

While POS (see above) exposed its interfaces to the client, it was the consensus of the ORBOS task force and the AB (Architecture Board) that PSS2 should be transparent to the client, and instead be written to serve the implementer of objects. This programmer, when creating his objects, some of which he wants to continue to maintain their own states persistently, may use the PSS2 implementation to help him achieve that persistent storage. Just as with Object Technology in general, the objects that use the PSS2 do so within their encapsulated implementations, hiding the details from their users (clients). Similarly, the PSS2, as defined in this specification, also takes on much, in fact almost all, of the drudgery of maintaining persistent state. It does expose enough interface to allow the object to manage its own persistent state, but also provides this interface in such a way that there is minimal burden on the programmer. Programmers can then make the decisions they wish, such as which objects will maintain persistent state, but leave the details to the PSS2, which comes close to transparently managing persistence for the programmer.

4.3. State

What is persistent? In the change from POS to PSS2, the name was explicitly changed in order to emphasize a different point of view. Until recently, OMG objects lacked any concept of state. The OMG object model defines objects as having interfaces, whose syntax (calling parameters, etc.) are well defined in IDL, and presumably these objects also have well-defined semantics for the operations invoked by such interfaces. Nowhere is there a concept of state.

For terminology, we follow the PSS2 RFP and use the term PS Object to denote an object whose state is being retained persistently. Note that PS Object refers to the object itself, not just the state that is being retained persistently.

Again, until recently, the OMG object continued to live wherever it wished to live, with location transparent to the user, and all implementation details (including any temporarily stored values or state, if the implementer desired) hidden from the client. The client simply invokes an operation (sends a request or message), which the ORB routes to the object, and the object executes. The concept of an object is that of an executing entity that implements operations, much as application servers often do.

Recently, with the Objects-by-Value service, for the first time OMG began to deal with the issue of state. One of the stronger responses to that RFP (doc. ORBOS/97-11-1, *from: IBM, Netscape, Novell and Visigenic*) includes a mechanism for the object to define its state, mostly independent of its interfaces, and then provides a mechanism for that state to be passed, via the ORB, to another location. There's no attempt to co-ordinate the passed value with the value being used within the object; rather, it uses copy semantics to take a snapshot of the desired values and send them off.

The use of state, for Objects-by-Value, is, in general, different from the purpose of this PSS2 RFP. In Objects-by-Value, the purpose is to take a snapshot of some relevant information which the object desires to export, then pass it to another object or client. Here, the purpose is to allow the object to maintain its own internal (encapsulated, hidden) state information, with enough retained persistently that the object can continue execution, consistently, at some later time, accessed by different users.

What we find in common with the Objects-by-Value submission, is the need to specify the state description of the object in IDL. It would be prudent (and possibly required by the OMG Architectural Board) for this submission to use the same IDL mechanisms rather than introduce redundant ones. That is our approach.

Briefly, the Objects-by-Value submission adds a new type to IDL, “value,” which complements the “interface” type, which specifies only interfaces. Value types provide semantics that support the description of complex state (i.e. arbitrary graphs, with recursion and cycles) that bridge between CORBA structs and CORBA interfaces. They support both public and private (to the implementation) data members. They support single inheritance (of value) and multiple inheritance of interface.

An example:

```
interface Employee {
      attribute   date  birthdate;
      attribute   string      name;

      void  fire();
      void  raise_salary();
};

value EmployeeV: Employee     {
      state {
            date  birthdate;
            string      name;
            long        salary;
      }
};
```

This allows declaring the internal state variables independently of the “attributes,” giving flexibility to the implementer to present whatever “attributes” he wish in IDL, while implementing them with whatever (the same or different) storage variables are desired.

So, there is a need in this response to allow the object implementer to define what state information s/he desires to preserve in order to allow the object to continue to operate persistently, with consistent behavior. We are also looking at requirements for schema definition that may go beyond what we expect to be in Objects-by-Value.

We are discussing the possible inclusion of relationships in the revised submission. For example, the current ODMG, Object Definition Language (ODL) allows you to specify relationships, which are not covered in the current Objects-by-Value submissions. There are also SQL3 proposals for ORDBMS DDL which would allow the direct use of the DBMS capability to support relationships and their integrity. This area is an open one for this initial submission and one we will revisit as we progress.

4.4. Identity

Until recently, OMG standards lack a specific concept of Object Identity. Identity means that there is a unique means to locate and access an object, independent of the object's state and location.

A way of addressing the uniqueness requirement is to use an identity-based object reference (often denoted object id, or OID), and a means to compare two different such OIDs to determine if they reference the same object. Today CORBA heavily uses the concept of object references, however there is no capability to determine if two object references refer to the same object.

One reason for this decision to omit identity is that, in general, in a distributed system, it may be difficult to implement. If objects appear and disappear, register freely with ORBs, without any central control source, with unlimited distribution, it becomes difficult to see how such a general identity mechanism can be supported. Without any central authority to coordinate OIDs, it is difficult to implement identity mechanisms. On the other hand, every CORBA implementation has just such a central coordinator (for dispatching, managing type repository, etc.) which can directly manage OIDs, or delegate it to other systems, while coordinating their individual OID ranges or namespaces. Today this is not done in a standards based way.

With the recent adoption of the ORB Portability Enhancement specification and it’s associated Portable Object Adapter (POA), some basic mechanisms for object identity are now introduced.

POA is designed to meet the following goals:

- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementation that can provide consistent service for objects whose lifetime span multiple server lifetime.
- Allow a single servant to support multiple object identities simultaneously.
- Allow object implementations to be maximally responsible for an object’s behavior. Specifically, an implementation can control an object’s behavior by establishing the datum that defines an objects identity, determining the relationship between that object’s identity and the objects state, managing the storage and retrieval of the object’s state, managing the storage and retrieval of the object’s state identity, determining the relationship between that object’s identity and the objects state, managing the storage and the retrieval of the object’s state providing the code that will be executed in response to requests and determining whether or not the object exists at any point in time.

- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities where their state is stored, whether certain identity values have been previously been used or not, and so on.
- Allow programmer to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compiler or a DSI implementation.The programmer's could use activation on demand mechanism to locate and retrieve the state from database and construct a servant of appropriate implementation class, initialize it with the state from the database and return it to the POA.
- The Explicit activation with POA assigned Object Id's mechanism could also be used: In this case, the developer could use some kind of database lookup method and find the persistent object associated with the servant class.
- The POA specifies how a CORBA object reference would encapsulate an Object ID and a POA identity. The POA deals with the registration of objects in a portable way. An Active Object Map is maintained within each POA for this purpose.

The approach we are suggesting would be a specific POA and it's associated servant object working with the PSS to manage the definition (format) and range of Object Ids that it knows about. For example, if database keys are used for persistent state retrieval, a POA servant using the PSS, would know how to use the Object Id to locate the corresponding persistent state in the data store. The PSS uses the key to fetch the corresponding information out of a database and initialize the servant object and its fields with the database data, before execution of the programmer's code begins. The key is also used later when the database is stored back on transaction commit.

ORB Portability and PSS work together to manage identity.

4.5. Technologies Addressed

The mechanism to retain the state of a PS Object may use a wide variety of technology, and different uses and users may have different requirements that are better met by a different implementations. To support these requirements, the PSS2 RFP specifically calls for a specification that will allow implementations in multiple technologies. This specification meets that need by supporting a very wide variety of technologies, but also meets the fundamental OMG requirement of interoperability by defining a single interface to be supported by all such technologies.

The model and architecture sections below describe how this is accomplished. The approach has been implemented in production systems that have been available for many years. These include systems that support this specification's interface for Relational Database Management Systems (RDBMSs), for Object-Relational Database Management Systems (ORDBMS), for Object Database Management Systems (ODBMSs), and even for flat file systems. Of course, depending on the technology and how close it is to supporting objects, implementers may need to layer more on top of the pre-object or non-object persistent storage system in order to meet this single object interface. This same holds true for all other CORBAservices (e.g., query servers, transaction servers, and even dispatchers or ORBs themselves).

4.6. Standards Support

OMG's goal of interoperability is achieved via CORBA and related facilities, all of which have well-defined interfaces, independent of any particular implementation. Where there are related standards already established and in-use, OMG has always sought to address them in some way,

if not adopt them specifically. This simply allows a wider range of interoperability by supporting a wider variety of existing products, tools, and systems, all of which may already be using some of these standards. Examples include IIOP and the Internet standards, CORBA and DCE, the Object Transaction Service (OTS) and X/Open's X/A protocol, the Object Query Service (OQS) and both SQL and OQL (the latter from ODMG), etc.

By following the same course, this specification supports a much wider variety of existing products and tools. Business users that wish to maintain their information persistently often find that the information they're maintaining is very critical to their business. For this reason, they often choose Database Management Systems (DBMSs), which are built specifically to maintain persistent information, but also to do so reliably, with the safety of recovery from failures, backup, archive, support for continued scalability as business systems grow, etc. Such DBMSs have adopted standards, including one of the most successful standards in computer technology, SQL (ref #), and the related call-level interface, ODBC (ref #). Similarly, over the last decade as DBMSs were created or evolved to better support objects, the Object Database Management Group (ODMG), working in cooperation with OMG, has designed a set of standard interfaces specifically for the purpose of maintaining the persistent state of objects.

This response specifically takes into account three major and widely-used standards:

- *OMG, including IDL, CORBA, and CORBAservices (as one would expect)*
- *ODBC and SQL, widely used not only in DBMSs, but also in tools*
- *ODMG, industry-accepted direct object interfaces*

Of course, any OMG adopted specification must support the OMG standards, as appropriate. The motivations for supporting SQL and ODBC include:

- *SQL is the most widely used, common interface to RDBMSs, and even some other systems such as ODBMSs. It provides a powerful query mechanism. SQL is an accredited international standard (from ISO) and is currently evolving to add support for "oo-ness" (reference, SQL3, ANSI and ISO) added to tables, the fundamental SQL and RDBMS structure. Supporting it allows immediately taking advantage of a large number of existing products, tools, and systems.*
- *ODBC, defined by Microsoft, is roughly based on the SQL Call-Level Interface. It allows embedded programmatic access to all the SQL-level functionality in a back-end DBMS. Further, it allows front-end tools to be implemented so that they work with any ODBC-compliant back-end DBMS. Tools can even be implemented, using ODBC, to simultaneously access multiple different back-end DBMSs. ODBC, then, adds support for the large number of existing tools, as well as programmatic access to a large number of back-end storage systems of various technologies, RDBMS, ODBMS, ORDBMS, and even some flat file systems.*

The motivations for supporting ODMG include:

- *The entire industry of ODBMSs, or databases designed to support objects, has unified behind this set of standard interfaces. This includes vendors of such ODBMSs, as well as a variety of end users, systems integrators, tools vendors, and even major system suppliers.*
- *The ODMG standard is mature. First released and quite useable in 1993 as V1.0, it has since been updated to V1.1, then V1.2, and in 1997 to V2.0. Each update is based on experience of implementations and users, providing improvements supporting increased functionality and improved interfaces, as well as supporting changes in related standards, such as object languages (C++ changes, appearance of Java, etc.).*
- *There are tried and proven implementations of this standard, as described in section 6.1.*

5. OVERVIEW OF MODEL AND ARCHITECTURE

5.1. Background

We now turn to address how this specification works, at a high level, first from the point of view of the model seen by its users (programmers), and next from the point of view of the major architecture concepts.

The user of PSS will see a simple model, as illustrated in figure 1.

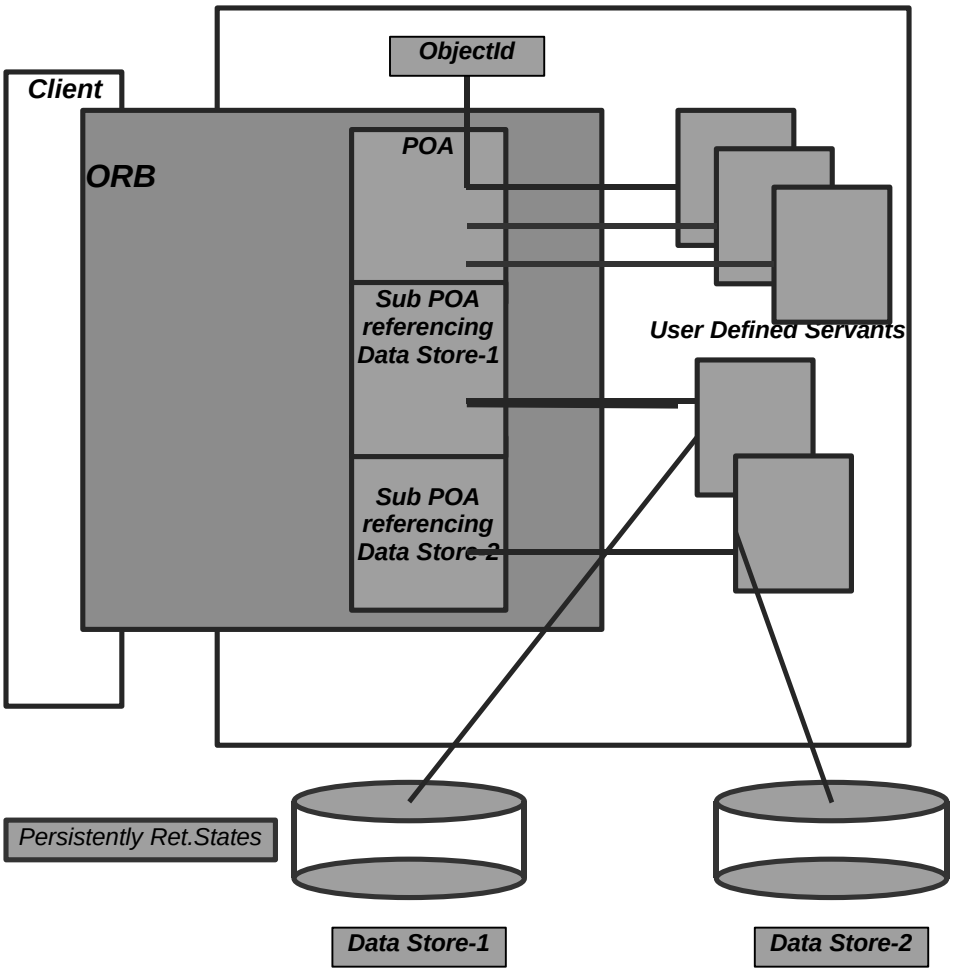


Fig 1. Single-Level Model

The client has no direct interface to the PSS and no need to invoke it. Rather, he simply benefits from the fact that some of the objects s/he's using do survive persistently.

The implementer of a PS object uses the PSS implementation to accomplish the actual work of storage of the persistent state. The PSS provides interfaces for the PS object implementer to allow him to access the PSS facilities, to define his own internal state, to allow the PSS to store that state for him as it changes, and to restore it, as needed. The programmer generally sees a clean, single-level store view of his object state. That is, the PSS maintains the links and coherency between the in-memory and on-disk images of the object's state, keeps them in sync, and manages the reading and writing as necessary.

In addition, the model as seen by the programmer includes the following:[1]

- *Client*—A client is a computational context that makes requests on an object through one of its references.
- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface. An object with persistent state in the PSS is called a PS Object.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.
- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.
- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.
- *POA*---A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.
- *Identity*---This is automatically managed by the PSS implementation so that the PS Object always obtains access to the correct state image. This is accomplished via use of a unique object identifier (OID) for each object or a mechanism to use database keys for a similar

[1] Some of these definitions have been repeated from the ORB Portability submission listed in Section 8 of this submission.

purpose. The OID is defined automatically by the PSS, provided for use by the object, and also used internally by the PSS to enforce correct behavior, connecting the correct persistent state information to the correct object. This OID is embedded opaquely within the OMG objref, as defined in POA, allowing each implementation to specify its own OID, and still provide interoperability. In other words, the PSS implementation defines what the OID is and provides access to it within the objref, but the actual implementation of the OID (including length, bit layout, etc.) is opaque (not visible to) the programmer. Where appropriate, database keys may be used in lieu of OIDs.

- *Transparent, Single-Level Persistence*---Unlike POS, the user of PSS2 sees only one state for his persistent object. The PSS implementation automatically manages any necessary work to copy images of that state from various physical locations, including disk and memory and whatever other caching mechanisms are used by the PSS implementation.
- *Automatic Activation and Access*---Typical OMG objects require, with both BOA and its replacement POA, a specific invocation to activate them, before accessing them. This scheme must be augmented to allow the transparent single-level store view described above. Instead, the PSS transparently provides access to the PS objects. Although it is necessary to activate the PSS itself, once that is done, it maintains access to and automatically activates as necessary all the PS Objects. This is important in practice not only for the transparent model, but also for performance and scalability. As the number of PS Objects grows very large, any need to independently register each one and activate it, requiring several ORB calls each, would quickly become prohibitive.
- *Integration with Other Services*---PSS implementations may transparently and automatically choose to provide other services integrated in, so they are automatically available for the PS Objects. This is common in DBMSs, which are, after file systems, the most heavily used persistent storage mechanisms, and more importantly, are use for the most critical business information. This includes capabilities such as transactions, concurrency control, security, as well as other often-desired capabilities including recovery, caching, clustering, distribution of object storage and execution locations, primitive relationships, replication, and fault tolerance.

5.2. Architecture

Supporting interoperability at the highest level requires a single interface. Anything short of that limits interoperability. This specification is based on a single interface that accomplishes this high level of interoperability. A single description language, IDL with objects-by-value extensions, is used to describe the persistent stored data. The binding of IDL to each language determines the programming language objects that will be materialized for the PS object programmer. These features allow the programmer writing the PS object to work entirely within the programming language with which they are familiar.

Finally, all of this is achieved within a layered architecture that allows a wide variety of implementations and architectures underneath. Of course, since the interface is based on objects, there is an advantage to systems that already include support for objects, but any other system can also support these interfaces simply by translating them to whatever primitives are inherent in the native technology of those other systems.

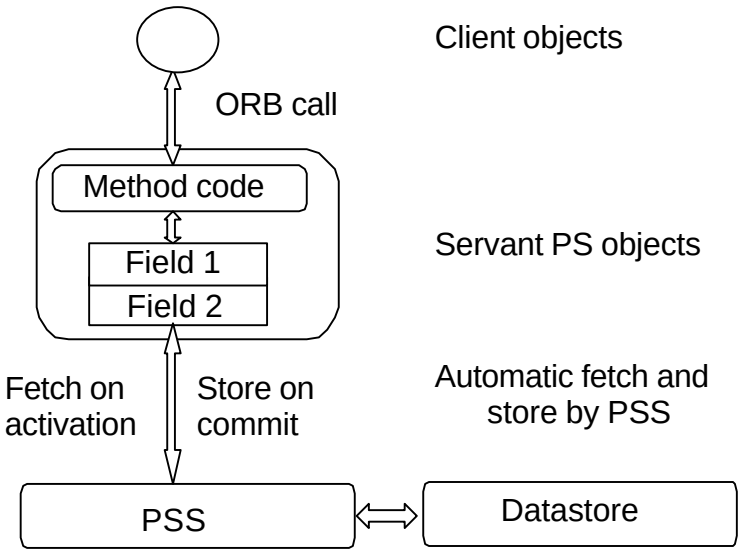


Fig 2. Persistence of CORBA servant objects

In figure 2 we illustrate how calls to servant objects automatically invoke the PSS to fetch and store data. The PS objects in the middle might be an application server or much simpler objects, or any OMG object which requires persistent state storage. The PS object programmer sees programming language persistence at runtime as defined in the appendices of this document. Transparent persistence allows the programmer to work naturally in his object language. Instead of making explicit invocations or function calls to copy data in and out of objects, the programmer simply uses the objects in the programming language, much as he uses ordinary transient objects.

The PSS can be very different in different implementations as illustrated in Figure 3. In an ODBMS it may simply be the DBMS itself, since it was already defined to support direct object requests. It might reside locally with the application, or at least partly so, to manage a local cache. In case of an ORDBMS with object cache, then it, too, may use its own object support directly as the object manager. For technologies, such as RDBMSs or flat files, a PSS2 compliant vendor would explicitly add a layer to translate objects into the underlying storage model. For example, for an RDBMS the PSS would use an object/relational mapping to translate object fetches and stores to operations such as SELECT, UPDATE, etc., on tables, as shown in the center of the figure, while for a flat file based system it would translate them to operations such as SEEK, READ, WRITE, on structures such as byte-array files or even indexed files. We look at object data mapping products in the next section.

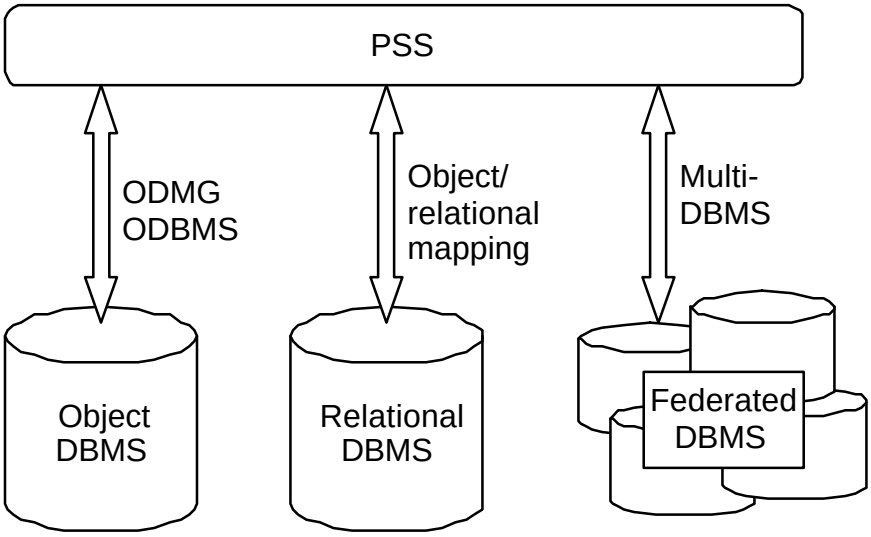


Fig. 3. Example PSS object managers

On the right side of the figure, we see that a PSS implementation may also support a federated DBMS that incorporates multiple data models, multiple DBMSs, and multiple databases on different machines.

Using the recent ORB Portability changes to CORBA, the PSS will take advantage of features offered by the POA specification. This allows an object reference (IOR) to encapsulate information that would help a PSS locate state information in the underlying database or file that correspond to a CORBA object. For example, in a relational DBMS, an IOR could encapsulate information such as the primary key (possibly encrypted) or rowid for the table in which objects of the given type are stored; for a file system it may store the file name; for an object DBMS and some ORDBMS products, it would simply store the database's OID. In the multi-database case on the right in figure 3, the OID would need to indicate which database has been used to store the PS object and then provide a locator within that database.

The implementers of objects decide which PSS-compliant storage system to use, but the clients see a common interface.

In short, the PSS provides a layer of abstraction, based on objects, which supports the common interface and provides interoperability subject to specifics of the data store. We specify only these few requirements on the PSS in order to allow the widest range of implementations underneath, and yet support portability and interoperability. The PSS integrates with the ORB to provide convenient activation of persistent objects using the mechanisms provided by the POA. The persistent data schema will be described in IDL using the proposed "Object By Value" extensions which is described in the following section.

5.3. State Definition

IDL with objects-by-value extensions is used to describe the persistent stored data. This state definition which can be created in one of two ways:

- The programmer can write persistent state definitions directly in IDL, and a parser (executed against this IDL or the Interface Repository) can generate base class implementations of the persistent state portion of the objects and their factories. In this case, the programmer would derive his servant implementation from this base class. This inheritance gives the programmer a consistent API for managing the persistent state of the object as well as registration of servants and factories with the PSS servant managers.
- Alternatively, the IDL definitions can be generated from programmer's class definitions automatically, allowing the programmer to work entirely within the programming language model. The use of this approach for C++, Smalltalk, and Java is illustrated in the appendices of this document, which are extracted from the ODMG 2.0 specification (reference 3). ODMG to date has used IDL *attributes* to define state; with the acceptance of the objects-by-value extensions and this PSS2 proposal, we propose that ODMG accept the state declaration syntax described here.

Below is an example of the first case (mapped to C++):

```
// IDL – borrowing the syntax of the Objects by Value submission
interface Employee {
        state {
                string          ssn;
                string          first_name;
                string          last_name;
                string          address;
                string          start_date;
                string          end_date;
        };
        void hire(in string ssn, in string first_name, in string last_name, in
                string address, in string start_date) raises (AlreadyEmployed);
        void fire(in string ssn, string end_date);
};
```

This definition gets mapped to a base class derived from the skeleton on the Server side:

```
// Generated C++
class PSSEmployee :     virtual public CosPersistent::PersistentObject,{
public:
        // Obligatory functions
                                PSSEmployee();
        virtual                 ~PSSEmployee();
        ...
        char *          ssn;
        char *          first_name;
        char *          last_name;
        char *          address;
        char *          start_date;
        char *          end_date;
        virtual void    ssn(const char *  ssn);

        virtual char *  ssn();

        virtual void    first_name(const char * fname);

        virtual char *  first_name();

        virtual void    last_name(const char * lname);
```

```
        virtual char *      last_name();
        virtual void        address(const char * addr);
        virtual char *      address();
        virtual void        start_date(const char * st_date);
        virtual char *      start_date();
        virtual void        end_date(const char * e_date);
        virtual char *      end_date();
};
```

Note: The current objects-by-value submission defines “getter” and “setter” methods for the C++ binding, as shown above. This is different from the current ODMG C++ binding as defined in Appendix B. We are currently inclined toward extending ODMG with the “getter” and “setter” methods. We will choose one or the other for the final submission. This incompatibility between ODMG and objects-by-value does not arise for Java, where both use fields, and Smalltalk, where both use methods.

The above class potentially owns its own state and connections to the database and manages its own persistence. The server programmer could then either derive from this class (and the skeleton) to write his implementation of the functions such as **hire()** and **fire()** or simply derive from the skeleton as usual and own the PSS generated class (or perhaps many of them).

```
class MyEmployeeImpl
        :       virtual public PSSEmployee,
        virtual public Employee_poa_impl
{
public:
        //ctors and dtors and other obligatory functions
        virtual void        hire(const char * ssn, const char *, fname, const char * lname,
                            const char * addr, const char * sdate);
        virtual void fire(const char * ssn, const char * edate);
};
```

Above we’ve inherited the persistent behavior. Alternatively the programmer can write:

```
class MyEmployeeImpl : virtual public Employee_poa_impl
{
public:
        //ctors and dtors and other obligatory functions
        virtual void        hire(const char * ssn, const char *, fname, const char * lname,
                            const char * addr, const char * sdate);
        virtual void fire(const char * ssn, const char * edate);

protected:
        PSSEmployee_ptr         m_pEmp;

};
```

In the above, the programmer delegates calls to a PSS generated class and could own several, if required to and manage them all.

A factory for this class would also be generated and derived from PersistentObjectFactory. The programmer could then implement a derived version of the factory overloading the create_object() operation to instantiate the objects however he/she sees fit.. The factory must also register itself with the PSS_ServantManager. This is a special servant manager that is given to the POA to manage the lifecycle of persistent objects and is supplied by the PSS implementer. The PSS_ServantManager will have to be derived from the ServantManager native interface described in the POA spec.

6. IMPLEMENTATION

This design is based on the combined experience of: major database vendors; including relational, non-relational and ODMG vendors, as well as database tool suppliers and ORB vendors.

6.1. Implementation Examples

As an example of the range of implementations that have already been shipped, we list the following samples. These are not exhaustive.

- Most ODBMSs have implemented at least some of these direct bindings, some with included support for ODL and/or OQL, and some with experience actually integrating with a CORBA-compliant product. Based on this experience, we are able to generalize to something that works and can be standardized. The ODBMSs include: Gemstone, Objectivity, Object Design, Poet, and Versant. Integrations with existing ORB products include several with the IONA Orbix product, as well as some with Visigenic, Expersoft, etc.
- Relational and Object/Relational Vendors have been providing a variety of products for binding objects to relational and object/relational data stores. Oracle and IBM have product experience with many of the technologies that are contained in this submission.
- RDBMSs have also been supported by this type of interface using object/relational mappings. Persistence Software, Ontos, and others have had extensive experience with a C++ direct language binding on top of several of the most popular RDBMSs, including Oracle, Sybase, and Informix. GemStone has had experience with a Smalltalk mapping to Sybase. Sun and O2 have recently announced ODMG-compliant mappings for Java running on JDBC-compliant DBMSs. Other companies with object/relational mapping products include HP, IBM, NeXT/Apple, Novera, Object Design, POET, and Subtle Software.
- Some Object/Relational DBMSs have implemented these interfaces, including direct language binding. An example is UniSQL.

6.2. Object Data Mapping

The object data mapping products automatically map data in an object to persistent data in a database or file. For example, an object/relational mapping product would create an Employee class definition in the Java programming language for the Employee table in a database:

```
class Employee {
    int empID;
    String name;
    Employee mgr,
    Department dept };
```

EMPID	NAME	MGR	DEPT

At runtime, the fields of Employee objects are automatically populated from the corresponding database rows in the Employee table, and are automatically stored back at the end of a transaction. Many products provide more sophisticated mappings between objects and databases, for example allowing a class to be the join of multiple tables, allowing one table to be partitioned into multiple classes, and automatically recognizing relationships and inheritance in the relational schema and mapping these to corresponding programming language data definitions. Database table definitions can be created automatically from existing classes, or class definitions can be created automatically from existing tables.

The original POS specification did not specify how the persistence service should integrate with an ORB, but with the advent of ORB portability and the POA this is now possible. We believe that this is the first feature a programmer wants in the PSS: the treatment of persistent state should be as convenient as possible without getting in the way of rich database functionality.

6.3. Managing Many Objects Efficiently

There is an efficiency issue when it is desired to connect many objects (say, 10 Million) to an ORB and make them accessible to the ORB client. In a simplistic approach, one would register each object with the ORB individually, a process which involves several ORB calls, which would be very slow and consume quite a bit of resources (memory for registration tables, etc.) for this many objects. This would not be an efficient approach and would not be effectively utilizing the facilities of the DBMS.

To avoid this huge overhead, we need mechanism to allow registering an object manager, which, in one registration dialogue with the ORB, explains to the ORB that it is responsible for some large number (or range) of objects. The ORB then knows to send all messages for any of those objects to that object manager. The mechanism allows implementers to provide facilities that will selectively choose which objects (or collections, containers, etc.) to register and thereby make visible through the ORB, while keeping others hidden and private.

Existing CORBA-compliant implementations have solved this problem in similar ways; e.g., Iona's Orbix adds the concept of a "Server," which may register with the ORB and tell the ORB it is responsible for several objects. The ORB then sends messages for any of those objects to the

Server, which maintains a table of objects it supports, and forwards messages to objects within that table (returning others as unknown). Similarly, Sun's NEO uses the concept of sub-objects.

Our approach to accomplish this starts with the newly adopted POA, a replacement for BOA and the means for an object implementer to register his object with the ORB. The "P" for "Portable" indicates that this registration mechanism can be implemented (by the object implementer) once and will work on any CORBA-compliant products.

POA adds a multiple object registration mechanism, including the concept of an object manager, to whom the ORB dispatches all requests for all objects in that object manager's domain. The object manager takes responsibility for dispatching to each of its own objects The implementer of the object manager (and the PSS) can then use this capability to register once with the ORB, and the ORB will know to send messages for all the objects controlled by that one object manager to that object manager.

Internally, this is done by embedding information in the objref. The POA specification already includes (reference 1) the ability for object implementers to embed, with their objrefs, an internal identification mechanism (such as an OID or primary key). This new POA subtype allows the CORBA implementation to access the high-order portion of this embedded identifier, and use that portion to identify which object manager is responsible, so it can forward requests to that object manager.

Note that we describe this enhancement to POA in terms of a general object manager. Although it is certainly critical for attaching DBMSs or any large repositories of objects, it can be useful in any other situation in which one object wants to be responsible for dispatching messages to some limited range of other objects.

Other ways to accomplish this were considered. For example, it is possible to create proxy objects on the fly, each of which dispatches (from the ORB) directly to the object manager. However, even if we start with only a small number of proxies (e.g., one per database or collection or...), the proliferation of millions of such proxies seems likely for robust users that access so many objects. We chose this approach not only for better efficiency, but in order to isolate, as much as possible, implementation from interface. Rather than imposing an implementation approach on the object manager and/or orb, we simple define an interface to allow the ORB and object manager to agree on an identifier (for the object manager).

This allows the ORB to immediately dispatch to that object manager, with minimal overhead, and probably with very little implementation effort; simply check for this object manager capability, and if appropriate, use that object for the target of the message. Also, it allows the implementer of the object manager (and PSS) to use any desired internal mechanism for identifying objects (rather than forcing, e.g., a table lookup). Object manager implementations that intend to scale to support many objects (such as DBMSs) must already have a scalable, efficient look up mechanism to access objects even as their number grows very large. With this approach, they're free to use exactly that same mechanism, and to embed within the objref whatever information, of whatever length, they need to accomplish that. All of this implementation is, of course, transparent to the client and to the ORB (and depending on the object manager implementation, it can also be transparent to the target object within the PSS).

This also works well with the Object Query Service. Queries can return (or reference) a huge number of objects. The OQS provides a simple and efficient mechanism to make this work, using the concept of collections, and iterators that can allow the query user to step through arbitrarily large numbers of returned objects in a scalable fashion that will not break simply because one query returns 10 objects while the next returns 10 million. The mechanism described here fits nicely into this OQS solution. It works well with collections, iterators, and objrefs and allows the ORB to understand the objrefs and dispatch to them. It allows PSS implementers the ability to directly use whatever internal query optimization techniques they have, e.g. a SQL query could be used to populate a large collection of persistent objects.

The following IDL is a description of the POA, as it relates to our usage here, and an example code fragment from a hypothetical implementer of an object manager, showing how to use this new registration mechanism. It includes a description mechanism to allow the ORB to identify the object manager, a means to create objrefs with the appropriate information inside them for this many-object dispatch mechanism, and a means for object implementers to extract that embedded information, if desired, for communicating directly with the object manager (DBMS or whatever).

6.4. Language Bindings

As introduced in the overview and architecture sections above, the PSS via the POA provides persistent programming language objects as CORBA servant PS objects. The semantics of persistence in the programming languages are defined by ODMG programming language bindings as specified in the appendices to this document. In general these ODMG APIs are “non-APIs”… that is, persistence is transparent in the programming language and the PS object programmer simply writes code in the language as they normally would with transient objects.

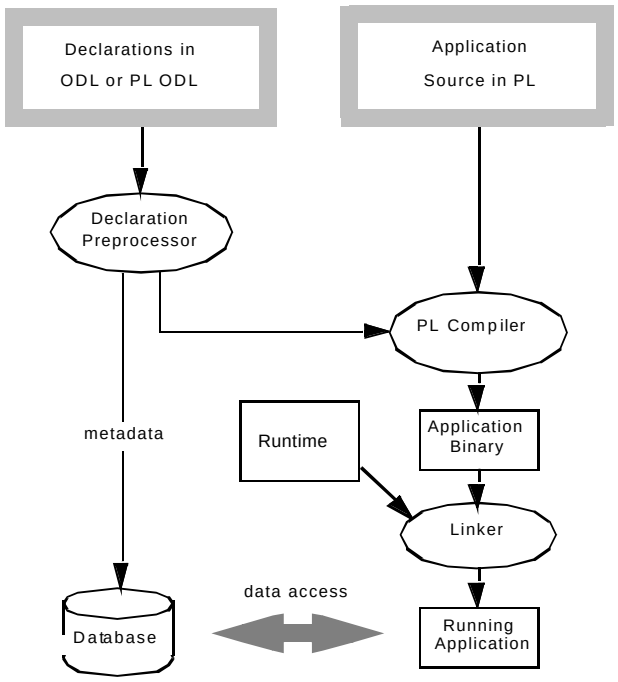
The language bindings are in ODMG 2.0 (reference #3), chapter 5, C++, chapter 6, Smalltalk, and chapter 7, Java. Each of these takes the ODMG object model, which is a superset of OMG's basic object model, and maps it to the host object language, using all the means available in each language to achieve the most transparent, natural to use interface, and one that is implementable with the highest efficiency. The model's primitives are mapped to the primitives of the host language, then the model’s higher-level concepts are mapped to existing facilities within the host language, when they exist. Otherwise, the model's concepts are mapped using the host language's inherent means to extend itself via operator overloading and added classes or types.

As a concrete example, ODMG provides for a means to reference an object with identity (d_Ref), both for accessing the object's own methods and for accessing relationships between such objects. In C++ this is mapped to a "smart pointer;" i.e., the C++ -> (arrow) operator is overloaded to automatically achieve the correct result. Exactly how it's overloaded is left to the implementation. Only the interfaces, the classes, where it's overloaded, where it's used, and the semantics of the overloading are actually defined. This allows, e.g., for one implementation to directly implement it as a raw pointer, using virtual memory, while another to implement it as a handle, providing an indirection mechanism which is transparent to the user, for additional safety, flexibility, and swapping capability. Many variations on this theme are possible.

The full text of these language bindings are included in the appendices. They are also available in ODMG 2.0. Should ODMG decide to modify these bindings in the future, this specification will continue to reference 2.0, for concreteness. It is expected that an OMG revision committee will track any important external changes, such as ODMG changes, language definition changes, etc., and choose to incorporate the appropriate changes into this OMG specification.

The figure below, extracted from ODMG 2.0, shows a typical programmer's view of the various ODMG interfaces. ODL (object definition language) is the ODMG extensions of IDL used for schema definition; as mentioned earlier, we propose to modify this for compatibility with the objects-by-value extensions to IDL. By defining his objects, both external interface (IDL) and state (the rest of ODL) with ODL, the programmer is provided with an automatic mapping to type or class definitions in each language. Then, the use of that language normally, along with any required ODMG-defined additional classes and operators, will produce the desired persistent behavior.

As shown in the figure, users may also choose to simply define their classes in the host programming language, using “PL ODL”. This is common where a customer uses only one programming language, for example. Of course, to support the IDL bindings, the user must then translate these to IDL. The specification of bindings show how to do this.



6.5. Related Services

Although it is theoretically possible to use a persistent storage mechanism by itself, in practice users often wish to simultaneously use various closely-related services, including OQS, OTS, etc. This specification takes these into account, providing close and effective integration with these other CORBAservices. This allows users who choose to use such related services, to immediately access products that have integrated support for multiple services in a consistent, coherent fashion, making them easy to use and highly efficient. In terms of quality of service, this specification allows use of storage technology, including DBMSs, that provide a wide range of quality, extending from simple, unprotected storage (file systems) through high-end DBMSs that support automatic recovery, back-up, archive, scalability, etc. The interfaces in this specification can be implemented on top of that full range, and have already been implemented on many of the major products.

6.5.1. Object Query Service

The QueryLanguageType IDL interface provides a very general model for OMG Query Service to be independent of any specific query language.

6.5.2. Object Lifecycle Service OLS

The OLS states that “The Life Cycle Service is not dependent on any particular model of persistence and is suitable for distributed, heterogeneous environments.” The PSS interface does not require any additional methods to support OLS and can be used with it as desired.

6.5.3. Object Concurrency Service OCS

The Concurrency Service provides interface (Locking) to mediate concurrent access to an object. The Concurrency Services provides interfaces which support transactional mode of operation. The PSS interface defined here supports the OCS.

6.5.4. Object Relationship Service ORS

The ORS works well with this specification and can be used as is. This allows relationship among PS Objects. Also, there is a lower-level, more primitive relationship mechanism included here for use by implementers of PS Objects that experience has shown to be quite useful. Also, implementers of ORS may choose to use these lower-level ODMG relationships as primitives out of which to build ORS relationships. This would simplify ORS implementation and allow it to leverage underlying systems (such as DBMSs with integrity support already built into relationships.)

6.5.5. Object Transaction Service OTS

The OTS works well with this specification and can be used as is at the level of coordinating multiple, independent resource managers. This allows PS Objects to inherit from the OTS Transaction context, in which case they'll automatically support this distributed transaction model.

6.6 Open Issues

6.6.1 Pre-Commit

In some situations, object implementers may need to exercise some control over the movement of data between object space and the underlying datastores. Object implementers would not have to explicitly "restore" data values when objects were accessed. Nor would they have to explicitly write back modified values at Commit. All of that activity would still be automatic. But for applications with more sophisticated semantics and datastores, object implementers could control some of the synchronization between object values and datastores. In particular, they could direct PSS to store back modified values before commit, or to re-access fresh values for objects that have already been activated. There are several scenarios involving relational databases that show the need for these optional controls. An object application may need to exploit complex triggers (side-effects) that are defined in the RDB. So if a new Customer Order triggers changes to Inventory and Backfill Order in the database, the same kinds of changes need to be visible in the corresponding objects. Some of these objects may already have been activated before the new Customer Object was created, and the application may re-access their attributes after the new Customer Object was created, all in the same transaction. So a write-back is needed to fire the trigger, and a refresh is needed to get the side-effects. Other examples are to do early checking of complex consistency constraints enforced by the database; to ensure that database stored procedures invoked by the application see the most recent values from changes made by that application; to ensure that the application sees the most recent side-effects of database stored procedures invoked by that same application; and to control sequencing of writebacks when database referential integrity constraints might otherwise be violated. The design for optional cache control is still an open discussion area.

6.6.2 Activate and Passivate

To achieve a finer grained control over the cache, an object implementer may want to use a passivate() hook, which notifies a client that an object he registered interest in has been passivated. The actual mechanics of how the object is serialized and finally swapped out is private to the object. Having, accomplished the above a client may still refer to the object that has been

passivated; this request results in the reclamation of the object and a notification sent to the client via the activate hook. Note that neither of these two activities can happen while an object is still in a transaction context, hence no data needs to be written out to persistent storage.

6.6.3 Other Fetch/Store Hooks

We are also considering hooks to allow alternatives to the transparent persistence binding to be used to fetch and store the PS object state, for example by invoking code for object serialization, file I/O, database calls, or embedded SQL. These hooks would obviate the need to build or use a transparent binding on top of these access mechanisms.

7. PERSISTENCE MODULE IDL

7.1. Overview

Client code accesses the persistence functionality via the CosPersistent Module. This section defines the various interfaces used here and describes the operations of these interfaces in detail

The following considerations were kept in mind when defining this interface:

- The interface for Object is referred to as PersistentObject, because OMG already uses this name for defining CORBA Objects. This interface could also inherit from CosTransaction::Resource interface, which would allow tighter integration of PSS with Transaction Service.
- The method s create_object (and create_db) are used to instantiate a persistent objects, the IDL compiler does not allow new to be a method in an interface.
- Concepts from the Objects-by-Value submission are used to provide semantics to support the description of complex state (i.e. arbitrary trees, graphs with recursion and cycles etc.) that bridge between CORBA structs and CORBA interfaces. Note: Objects-by-Value is not yet an adopted specification. We are planning to align with what is finally adopted since we expect this to happen before the PSS RFP process completes.
- There are currently a number of incompatible IDL specifications and models for collections (lists, sets, etc), including those defined in the Relationship Service, Query Service, Collection Service, Business Object Facility, and the ODMG Object Model. Rather than defining an IDL for persistent collections, we permit any collection implementation to be used. In addition, collection classes have been standardized in programming languages, so the language bindings in the appendices use variations of those standards in order to be natural to the programmer in each language.
- We do not currently require that a PSS implementation implement queries on datastore objects. However, for those that do, the appendices define a standard way to do so. In addition, the Object Query Service may be implemented on a datastore or on a number of CORBA objects and datastores.

```
module CosPersistent
{
    enum Lock_Type {read,write,upgrade};

    typedef unsigned long ulong;

    exception LockNotGranted{};
    exception InvalidCollectionType{};
    exception NoMoreElements{};
    exception InvalidIndex{unsigned long index;};
    exception ElementNotFound{any element;};
    exception KeyNotFound{any key;};
```

```
    interface PersistentObjectFactory;
    interface PersistentObject;

    interface DatabaseFactory;
    interface Database;
    struct Association {
        any key;
        any value;
    };
```

7.2. The PersistentObjectFactory interface

```
interface PersistentObjectFactory {
        PersistentObject create_object();
    };
```

This interface is used to create a new PersistentObject.

7.2.1. create_object

```
PersistentObject create_object();
```

This method creates a new PersistentObject and returns its reference.

7.3. The PersistentObject Interface

```
interface PersistentObject {
        void lock(in Lock_Type mode)
             raises(LockNotGranted);
        boolean try_lock(in Lock_Type mode);
        boolean same_as(in Object anObject);
        PersistentObject copy();
        void remove();
    };
```

The PersistentObject interface provides methods that can be used to manage the state of the object.

7.3.1. lock

```
void lock(in Lock_Type mode) raises (LockNotGranted);
```

This method acquires a lock on the PersistentObject in the specified mode. If another client holds a lock in the same object in an incompatible mode, the then operation will raise an exception LockNotGranted.

7.3.2. try_lock

```
boolean try_lock(in Lock_Type mode);
```

This method attempts to acquire a lock on the PersistentObject. If a lock is already held in an incompatible mode by another client then the operation returns a FALSE result to indicate that the lock could not be acquired.

7.3.3. same_as

```
boolean same_as(in Object anObject);
```

This method compares the PersistentObject with the specified object. If both the objects are same then TRUE is returned.

7.3.4. copy

```
PersistentObject copy();
```

This method is used to create a copy this PersistentObject.

7.3.5. remove

```
void remove();
```

This method is used to remove the PersistentObject from the datastore. This method would not remove the CORBA Object related to the PersistentObject.

7.4. The DatabaseFactory interface

```
interface DatabaseFactory {
        Database create_db();
    };
```

This interface is used to create a Database object.

7.4.1. create_db

```
Database create_db();
```

This method is used to create a new Database object.

7.5. The Transaction Interface

7.5.1. Transactions and Processes

The OMG CORBAServices includes an Object Transaction Service that works at the level of coordinating multiple, independent resource managers. The PS Objects may optionally inherit from the OTS Transaction Object, in which case they'll automatically support this distribution transaction model. In addition, even without OTS, the PSS supports a concept of transaction at one lower level, that of an individual resource manager for that datastore. This transaction interface specifies that lower level. The higher level, or distributed transaction level, is defined in the OTS.

We assume a linear sequence of transactions executing within a thread of control; that is, there is exactly one current transaction for a thread, and that transaction is implicit in that thread's database operations. If a language binding supports multiple threads in one address space, then transaction *isolation* must be provided between the threads. Of course, transaction *isolation* is also provided between threads in different address spaces or threads running on different machines.

A transaction runs against a single logical database. Note that a single logical database may be implemented as one or more physical databases, possibly distributed on a network. The transaction model neither requires nor precludes support for transactions that span multiple threads, multiple address spaces, or more than one logical database.

In the current model, transient objects in an address space are not subject to transaction semantics. This means that aborting a transaction does not restore the state of modified transient objects.

Two-level data stores or other datastores which wish to provide operations without transactional semantic and ACID properties (such as a file system or RDBMS or mainframe DBMS) the semantics of the transaction object operatoins may be rendered as null. The result is that the user's code is portable across datastores providing different levels of quality of service. On some, a file system for example, the invocation of start() and commit() is ignored, there is no recovery or concurrency control, so abort() is also a null operation. The same application code (OMG Object implementation), working with a higher quality of service datastore, will see the benefits of concurrency control, recovery, and functionality of abort(), along with full ACID properties. Some intermediate choices of quality of service would be to support the semantics of checkpoint (making changes visible to others and saved on disk) without the guarantees of ACID properties. Other qualities of service may be provided, as long as the provider specifies their semantics.

7.5.2. Transaction Operations

There are two types that are defined to support transaction activity within an ODBMS: TransactionFactory and Transaction.

The TransactionFactory type is used to create transactions. The following operations are defined in the TransactionFactory interface:

```
interface TransactionFactory {
      Transaction                    new();
```

```
      Transaction                   current();
};
```

The new operation creates Transaction objects. The current operation returns the Transaction that is associated with the current thread of control. If there is no such association, the current operation returns *nil*.

Once a Transaction object is created, it is manipulated using the Transaction interface. The following operations are defined in the Transaction interface:

```
interface Transaction {
      exception                     TransactionInProgress{};
      exception                     TransactionNotInProgress{};
      void                    begin()
raises(TransactionInProgress);
      void                    commit()
raises(TransactionNotInProgress);
      void                    abort()
raises(TransactionNotInProgress);
      void                    checkpoint()
raises(TransactionNotInProgress);
      void                    join();
      void                    leave();
      boolean                 isOpen();
};
```

After a Transaction object is created, it is initially closed. An explicit begin operation is required to open a transaction. If a transaction is already open, additional begin operations raise the TransactionInProgress exception.

The commit operation causes all persistent objects created or modified during a transaction to be written to the database and become accessible to other Transaction objects running against that database. All locks held by the Transaction object are released. Finally, it also causes the Transaction object to complete and become closed. The TransactionNotInProgress exception is raised if a commit operation is executed on a closed Transaction object.

The abort operation causes the Transaction object to complete and become closed. The database is returned to the state it was in prior to the beginning of the transaction. All locks held by the Transaction object are released. The TransactionNotInProgress exception is raised if an abort operation is executed on a closed Transaction object.

A checkpoint operation is equivalent to a commit operation followed by a begin operation, except that locks held by the Transaction object are NOT released. Therefore, it causes all modified objects to be committed to the database and it retains all locks held by the Transaction object. The Transaction object remains open. The TransactionNotInProgress exception is raised if a checkpoint operation is executed on a closed Transaction object.

As mentioned above, varying qualities of service may be offered in different implementations, some of which might choose to implement the above operations [start(), commit(), abort(), checkpoint(), etc.] as null operations, or operations with semantics less that complete serializability and ACID properties, as long as the provider of such a service specifies its quality level, including the semantics of these operations.

Checkpointing and QOS because of the varying degree of functionality and their sophistication are currently under discussion.

Database operations are always applied to the database during a transaction. Therefore, to execute any database operations, an active Transaction object must be associated with the current thread. The join operation associates the current thread with a Transaction object. If the

Transaction object is open, database operations may be executed; otherwise a TransactionNotInProgress exception is raised.

If an implementation allows multiple active Transaction objects to exist, the join and leave operations allow a thread to alternate between them. To associate the current thread with another Transaction object, simply execute a join on the new Transaction object. If necessary, a leave operation is automatically executed to disassociate the current thread from its current Transaction object. Moving from one Transaction object to another does not commit or abort a Transaction object. When the current thread has no current Transaction object, the leave operation is ignored.

After a Transaction object is completed, to continue executing database operations, either another open Transaction object must be associated with the current thread, or a begin operation must be applied to the current Transaction object to make it open again.

Multiple threads of control in one address space can share the same transaction through multiple join operations on the same Transaction object. In this case, no locking is provided between these threads; concurrency control must be provided by the user. The transaction completes when any one of the threads executes a commit or abort operation against the Transaction object.

7.6. The Database interface

```
interface Database {
        void open(in string database_name);
        void close();
        void bind(in any an_object,in string name);
        Object unbind(in string name);
        Object lookup(in string object_name);
        ModuleDef schema();
    };
```

Once a Database object is created by using the create_db operation, it is manipulated using the Database interface.

7.6.1. open

```
void open(in string database_name);
```

The open operation must be invoked, with a database name as its argument, before any access can be made to the persistent objects in the database.

7.6.2. close

```
void close();
```

The close operation must be invoked when a program has completed all access to the database.

7.6.3. lookup

```
Object lookup(in string object_name);
```

The lookup operation finds the identified of the object with the name supplicated as the argument to the operation. This operation is defined on the Database object, because the scope of object names is the database.

7.6.4. bind

```
void bind(in any an_object, in string name);
```

A name is bound to a Persistent Object using the bind operation.

7.6.5. unbind

```
Object unbind(in string name);
```

Named objects may be unnamed using the unbind operation.

7.6.6. schema

```
ModuleDef schema();
```

The schema operation accesses the meta data that defines the schema of the database.

8. DOCUMENT REFERENCES

1. OMG ORB Portability Joint Submission (Final), orbos/97-05-15

2. IBM, Netscape, Novell and Visigenic, Objects By Value Joint Revised Submission Draft Version 5.2

3. Cattell et al, The Object Database Standard: ODMG 2.0, Morgan Kaufman, 1997.

9. APPENDIX A - ODMG CHAPTER 1, OVERVIEW

Chapter 1

Overview

1.1 Background

This document describes the continuing work on standards for object database management systems (ODBMSs) undertaken by the members of the Object Database Management Group (ODMG). This specification represents an enhancement to ODMG-93, Release 1.2.

We have worked outside of traditional standards bodies for our efforts in order to make quick progress. Standards groups are well suited to incremental changes to a proposal once a good starting point has been established, but it is difficult to perform substantial creative work in such organizations due to their lack of continuity, large membership, and infrequent meetings. It should be noted that relational database standards started with a database model and language implemented by the largest company involved (IBM); for our work, we have picked and combined the best features of implementations we had available to us.

1.1.1 Importance of a Standard

Before ODMG, the lack of a standard for object databases was a major limitation to their more widespread use. The success of relational database systems did not result simply from a higher level of data independence and a simpler data model than previous systems. Much of their success came from the standardization that they offer. The acceptance of the SQL standard allows a high degree of portability and interoperability between systems, simplifies learning new relational DBMSs, and represents a wide endorsement of the relational approach.

All of these factors are important for object DBMSs, as well. The scope of object DBMSs is more far-reaching than that of relational DBMSs, integrating the programming language and database system, and encompassing all of an application's operations and data. A standard is critical to making such applications practical.

The intense ODMG effort has given the object database industry a "jump start" toward standards that would otherwise have taken many years. ODMG enables many vendors to support and endorse a common object database interface to which customers write their applications.

1.1.2 Goals

Our primary goal is to put forward a set of standards allowing an ODBMS customer to write portable applications, i.e., applications that could run on more than one ODBMS product. The data schema, programming language binding, and data manipulation and query languages must be portable. Eventually, we hope our standards proposal will be helpful in allowing interoperability between the ODBMS products as well, e.g., for heterogeneous distributed databases communicating through the OMG Object Request Broker.

We are striving to bring programming languages and database systems to a new level of integration, moving the industry forward as a whole through the practical impetus of real products that conform to a more comprehensive standard than is possible with relational systems. We have gone further than the least common denominator of the first relational standards, and we want to provide portability for the entire application, not just the small portion of the semantics encoded in embedded SQL statements.

The ODMG member companies, representing almost the entire ODBMS industry, are supporting this standard. Thus, our proposal has become a de facto standard for this industry. We have also used our specification in our work with standards groups such as the OMG and the ANSI X3H2 (SQL) committee.

We do not wish to produce identical ODBMS products. Our goal is source code portability; there is a lot of room for future innovation in a number of areas. There will be differences between products in performance, languages supported, functionality unique to particular market segments (e.g., version and configuration management), accompanying programming environments, application construction tools, small versus large scale, multithreading, networking, platform availability, depth of functionality, suites of predefined type libraries, GUI builders, design tools, and so on.

Wherever possible, we have used existing work as the basis for our proposals, from standards groups and from the literature. But, primarily, our work is derived by combining the strongest features of the ODBMS products currently available. These products offer demonstrated implementations of our standards components that have been tried in the field.

1.1.3 Definition

It is important to define the scope of our efforts, since ODBMSs provide an architecture that is significantly different than other DBMSs — they are a revolutionary rather than an evolutionary development. Rather than providing only a high-level language such as SQL for data manipulation, an ODBMS transparently integrates database capability with the application programming language. This transparency makes it unnecessary to learn a separate DML, obviates the need to explicitly copy and translate data between database and programming language representations, and supports substantial

performance advantages through data caching in applications. The ODBMS includes the query language capability of relational systems as well, and the query language model is more powerful; e.g., it incorporates lists, arrays, and results of any type. Figure 1-1 provides a comparison of DBMS architectures.

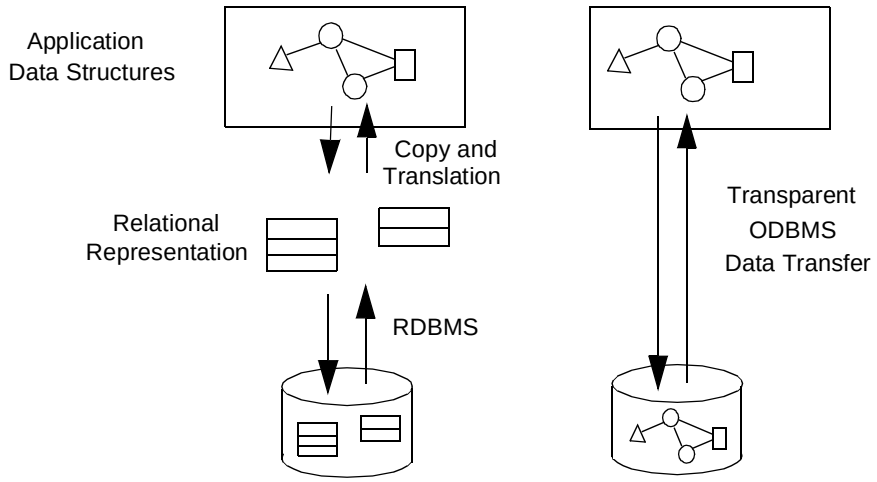


Figure 1-1. Comparison of DBMS Architectures

In summary, we define an *ODBMS* to be a DBMS that integrates database capabilities with object-oriented programming language capabilities. An ODBMS makes database objects appear as programming language objects, in one or more existing programming languages. The ODBMS extends the language with transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities. For more extensive definition and discussion of ODBMSs, the reader is referred to textbooks in this area (e.g., Cattell, *Object Data Management*).

1.2 Architecture

In order to understand the chapters of this book, it is necessary to understand the overall architecture of ODBMSs.

1.2.1 Major Components

The major components of ODMG 2.0 are described in subsequent chapters:

> *Object Model.* The common data model to be supported by ODBMSs is described in Chapter 2. We have used the OMG Object Model as the basis for

our model. The OMG core model was designed to be a common denominator for object request brokers, object database systems, object programming languages, and other applications. In keeping with the OMG Architecture, we have designed an ODBMS *profile* for their model, adding components (e.g., relationships) to the OMG core object model to support our needs. Release 2.0 introduces a meta model in this chapter.

Object Specification Languages. The specification languages for ODBMSs are described in Chapter 3. One is the object definition language, or ODL, to distinguish it from traditional database data definition languages, or DDLs. We use the OMG interface definition language (IDL) as the basis for ODL syntax. Release 2.0 adds another language, the object interchange format, or OIF, which can be used to exchange objects between databases, provide database documentation, or drive database test suites.

Object Query Language. We define a declarative (nonprocedural) language for querying and updating database objects. This object query language, or OQL, is described in Chapter 4. We have used the relational standard SQL as the basis for OQL, where possible, though OQL supports more powerful capabilities.

C++ Language Binding. Chapter 5 presents the standard binding of ODBMSs to C++; it explains how to write portable C++ code that manipulates persistent objects. This is called the C++ OML, or object manipulation language. The C++ binding also includes a version of the ODL that uses C++ syntax, a mechanism to invoke OQL, and procedures for operations on databases and transactions.

Smalltalk Language Binding. Chapter 6 presents the standard binding of ODBMSs to Smalltalk; it defines the binding in terms of the mapping between ODL and Smalltalk, which is based on the OMG Smalltalk binding for IDL. The Smalltalk binding also includes a mechanism to invoke OQL and procedures for operations on databases and transactions.

Java Language Binding. Chapter 7 defines the binding between the ODMG Object Model (ODL and OML) and the Java programming language as defined by Version 1.1 of the Java™ Language Specification. The Java language binding also includes a mechanism to invoke OQL and procedures for operations on databases and transactions.

It is possible to read and write the same database from C++, Smalltalk, and Java, as long as the programmer stays within the common subset of supported data types. More

chapters may be added at a future date for other language bindings. Note that unlike SQL in relational systems, ODBMS data manipulation languages are tailored to specific application programming languages, in order to provide a single, integrated environment for programming and data manipulation. We don't believe exclusively in a universal DML syntax. We go further than relational systems, as we support a unified object model for sharing data across programming languages, as well as a common query language.

1.2.2 Additional Components

In addition to the object database standards, ODMG has produced some ancillary results aimed at forwarding the ODBMS industry. These are included as appendices:

OMG Object Model Profile. Appendix A describes the differences between our object model and the OMG Object Model, so that Chapter 2 can stand alone. As just mentioned, we have defined the components in an ODBMS profile for OMG's model. This appendix delineates these components.

OMG ORB Binding. Appendix B describes how ODBMS objects could participate as OMG objects, through an adaptor to an object request broker (ORB) that routes object invocations through object identifiers provided by an ODBMS. We also outline how ODBMSs can make use of the OMG ORB.

1.2.3 ODBMS Architecture Perspective

A better understanding of the architecture of an ODBMS will help put the components we have discussed into perspective.

Figure 1-2 illustrates the use of the typical ODBMS product that we are trying to standardize. The programmer writes declarations for the application schema (both data and interfaces) plus a source program for the application implementation. The source program is written in a programming language (PL) such as C++, using a class library that provides full database OML, including transactions and object query. The schema declarations may be written in an extension of the programming language syntax, labeled PL ODL in the figure, or in a programming language–independent ODL. The latter could be used as a higher-level design language, or to allow schema definition independent of programming language.

The declarations and source program are then compiled and linked with the ODBMS to produce the running application. The application accesses a new or existing database, whose types must conform to the declarations. Databases may be shared with other applications on a network; the ODBMS provides a shared service for transaction and lock management, allowing data to be cached in the application.

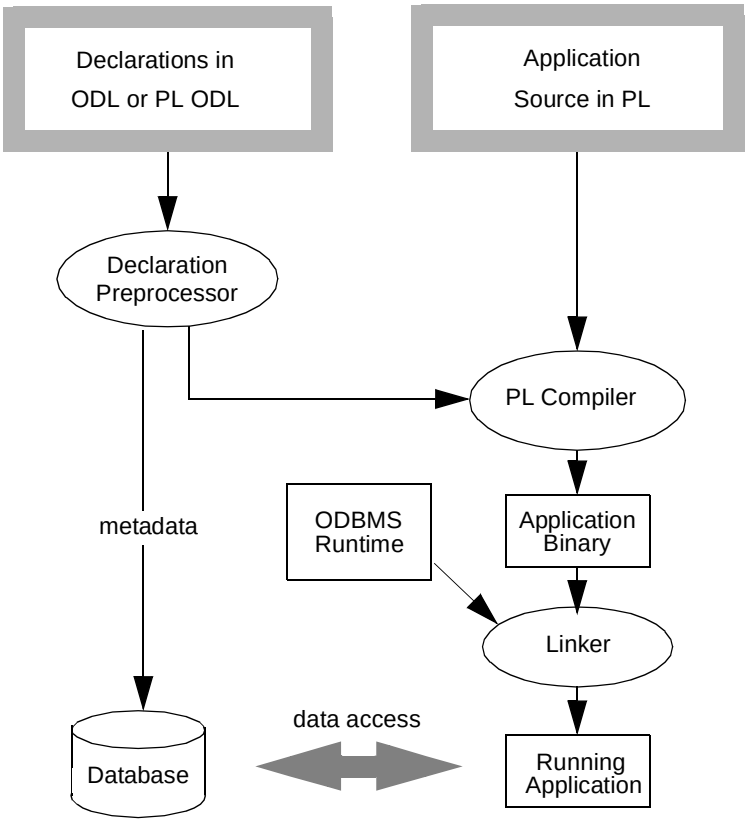


Figure 1-2. Using an ODBMS

1.3 Status

This document describes Release 2.0 of the ODMG standard. The ODMG voting member companies and many of the reviewer member companies are committed to support this standard in their products by the end of 1998.

1.3.1 Participants

As of March 1997, the participants in the ODMG are

- Rick Cattell (ODMG chair, Java workgroup chair, Release 1.0 editor), JavaSoft
- Jeff Eastman (ODMG vice-chair, object model workgroup chair, Smalltalk editor), Windward Solutions

- Douglas Barry (ODMG executive director, Release 1.1, 1.2, and 2.0 editor), Barry & Associates
- Mark Berler (object model and object specification languages editor), American Management Systems
- David Jordan (C++ editor), Lucent Technologies
- Francois Bancilhon, Sophie Gamerman (OQL editor), voting member, O_2 Technology
- Dirk Bartels (C++ workgroup chair), Olaf Schadow, voting member, POET Software
- William Kelly, voting member, UniSQL
- Paul Pazandak, voting member, IBEX
- Ken Sinclair, voting member, Object Design
- Adam Springer (Smalltalk workgroup chair), voting member, GemStone Systems
- Henry Strickland (Java editor), Craig Russell, voting member, Versant Object Technology
- Drew Wade (OQL workgroup chair), Jacob Butcher, Dann Treachler, voting member, Objectivity
- Michael Card, reviewer member, Lockheed Martin
- Edouard Duvillier, reviewer member, MATISSE Software
- Jean-Claude Franchitti, reviewer member, VMARK Software
- Marc Gille, reviewer member, MICRAM Object Technology
- William Herndon, reviewer member, MITRE
- Mamdouh Ibrahim, reviewer member, Electronic Data Systems
- Richard Jensen, reviewer member, Persistence Software
- Yutaka Kimura, reviewer member, NEC
- Shaun Marsh, reviewer member, Fujitsu Open Systems Solutions
- Richard Patterson, reviewer member, Microsoft
- Shirley Schneider, reviewer member, ONTOS
- Jamie Shiers, reviewer member, CERN
- John Shiner, reviewer member, Andersen Consulting
- Jacob Stein, reviewer member, Sybase
- Fernando Velez, reviewer member, Unidata
- Satoshi Wakayama, reviewer member, Hitachi

It is to the personal credit of all participants that the ODMG standard has been produced and revised expeditiously. All of the contributors put substantial time and personal investment into the meetings and this document. They showed remarkable

dedication to our goals; no one attempted to twist the process to his or her company's advantage. The reviewers were also very helpful, always ready to contribute.

In addition to the regular ODMG participants above, we received valuable feedback from others in academia and industry. We would like to thank our academic reviewers, in particular Eliot Moss for his contribution to the object model chapter. We would also like to thank Joshua Duhl for his exhaustive review of Release 1.2 as part of the process to create certification test suites.

1.3.2 History

Some of the history and methodology of ODMG may be helpful in understanding our work and the philosophy behind it. We learned a lot about how to make quick progress in standards in a new industry while avoiding "design by committee."

ODMG was conceived at the invitation of Rick Cattell in the summer of 1991, in an impromptu breakfast with ODBMS vendors frustrated at the lack of progress toward ODBMS standards. Our first meeting was at SunSoft in the fall of 1991.

The group adopted rules that have been instrumental to our quick progress. We wanted to remain small and focused in our work, yet be open to all parties who are interested in our work. The structure evolved over time. Presently, we have established workgroups, one for each chapter of the specification. Each workgroup is intended to remain small, allowing for good discussion. The specifications adopted in each workgroup, however, must go before the ODMG Board for final approval. The Board usually holds open meetings for representatives from all members to attend and comment on our work.

The people who come to our meetings from our member companies are called Technical Representatives. They are required to have a technical background in our industry. We also have established rules requiring the same Technical Representatives come repeatedly to our meetings to maintain continuity of our work. Technical Representatives from voting member companies often contribute 25 percent of their time to the ODMG. Some of the Technical Representatives from our reviewer members contribute an equal amount of time while others are more in the range of 10 percent of their time.

Voting membership is open to organizations that have developed and commercially sell a currently shipping ODBMS as defined on page 3. Reviewer members are individuals or organizations having a direct and material interest in the work of the ODMG. Certification members, our newest group of members, are individuals or organizations having a direct and material interest in the certification work of the ODMG.

1.3.3 Accomplishments

Since the publication of Release 1.0, a number of activities have occurred.

1. Incorporation of the ODMG and the establishment of an office.
2. Affiliation with the Object Management Group (OMG), OMG adoption (February 1994) of a Persistence Service endorsing ODMG-93 as a standard interface for storing persistent state, and OMG adoption (May 1995) of a Query Service endorsing the ODMG OQL for querying OMG objects.
3. Establishment of liaisons with ANSI X3H2 (SQL), X3J16 (C++), and X3J20 (Smalltalk), and ongoing work between ODMG and X3H2 for converging OQL and SQL3.
4. Addition of reviewer membership to allow the user community to participate more fully in the efforts of the ODMG.
5. Addition of certification membership to allow the user community to participate in the development of our certification test suites.
6. Publication of articles written by ODMG participants that explain the goals of the ODMG and how they will affect the industry.
7. Collection of feedback on Release 1.0, 1.1, and 1.2, of which much was used in this release.

1.3.4 Next Steps

We now plan to proceed with several actions in parallel to keep things moving quickly.

1. Distribute Release 2.0 through this book.
2. Complete implementation of the specifications in our respective products.
3. Collect feedback and corrections for the next release of our standards specification.
4. Continue to maintain and develop our work.
5. Continue to submit our work to OMG or ANSI groups, as appropriate.

1.3.5 Suggestion and Proposal Process

If you have suggestions for improvements in future versions of our document, we welcome your input. We recommend that change proposals be submitted as follows:

1. State the essence of your proposal.
2. Outline the motivation and any pros/cons for the change.
3. State exactly what edits should be made to the text, referring to page number, section number, and paragraph.
4. Send your proposal to proposal@odmg.org.

1.3.6 Contact Information

For more information about the ODMG and the latest status of its work, send electronic mail to info@odmg.org. You will receive an automated response.

If you have questions on ODMG 2.0, send them to question@odmg.org.

If you have additional questions, or if you want membership information for the ODMG, please contact ODMG's executive director, Douglas Barry, at dbarry@odmg.org, or contact:

Object Database Management Group
14041 Burnhaven Drive, Suite 105
Burnsville, MN 55337 USA
voice: +1-612-953-7250
fax: +1-612-397-7146
email: info@odmg.org
Web: www.odmg.org

1.3.7 Related Standards

There are references in this book to ANSI X3 documents, including SQL specifications (X3H2), Object Information Management (X3H7), the X3/SPARC/DBSSG OODB Task Group Report (contact fong@ecs.ncsl.nist.gov), and the C++ standard (X3J16). ANSI documents can be obtained from:

X3 Secretariat, CBEMA
1250 Eye Street, NW, Suite 200
Washington, DC 20005-3922 USA

There are also references to Object Management Group (OMG) specifications, from the Object Request Broker (ORB) Task Force (also called CORBA), the Object Model Task Force (OMTF), and the Object Services Task Force (OSTF). OMG can be contacted at:

Object Management Group
Framingham Corporate Center
492 Old Connecticut Path
Framingham, MA 01701 USA
voice: +1-508-820-4300
fax: +1-508-820-4303
email: omg@omg.org
Web: www.omg.org

10. APPENDIX B - ODMG CHAPTER 5, C++ BINDING

Chapter 5

C++ Binding

5.1 Introduction

This chapter defines the C++ binding for ODL/OML.

ODL stands for Object Definition Language. It is the declarative portion of C++ ODL/OML. The C++ binding of ODL is expressed as a library that provides classes and functions to implement the concepts defined in the ODMG Object Model. OML stands for Object Manipulation Language. It is the language used for retrieving objects from the database and modifying them. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library.

ODL/OML specifies only the logical characteristics of objects and the operations used to manipulate them. It does not discuss the physical storage of objects. It does not address the clustering or memory management issues associated with the stored physical representation of objects or access structures like indices used to accelerate object retrieval. In an ideal world these would be transparent to the programmer. In the real world they are not. An additional set of constructs called *physical pragmas* is defined to give the programmer some direct control over these issues, or at least to enable a programmer to provide "hints" to the storage management subsystem provided as part of the ODBMS runtime. Physical pragmas exist within the ODL and OML. They are added to object type definitions specified in ODL, expressed as OML operations, or shown as optional arguments to operations defined within OML. Because these pragmas are not in any sense a stand-alone language, but rather a set of constructs added to ODL/OML to address implementation issues, they are included within the relevant subsections of this chapter.

The chapter is organized as follows. Section 5.2 discusses the ODL. Section 5.3 discusses the OML. Section 5.4 discusses OQL—the distinguished subset of OML that supports associative retrieval. Associative retrieval is access based on the values of the properties of objects rather than on their IDs or names. Section 5.6 provides an example program.

5.1.1 Language Design Principles

The programming language–specific bindings for ODL/OML are based on one basic principle: The programmer feels that there is one language, not two separate languages with arbitrary boundaries between them. This principle has two corollaries that are evident in the design of the C++ binding defined in the body of this chapter:

1. There is a single unified type system across the programming language and the database; individual instances of these common types can be persistent or transient.
2. The programming language–specific binding for ODL/OML respects the syntax and semantics of the base programming language into which it is being inserted.

5.1.2 Language Binding

The C++ binding maps the Object Model into C++ by introducing a set of classes that can have both persistent and transient instances. These classes are informally referred to as "persistence-capable classes" in the body of this chapter. These classes are distinct from the normal classes defined by the C++ language, all of whose instances are transient; that is, they don't outlive the execution of the process in which they were created. Where it is necessary to distinguish between these two categories of classes, the former are called "persistence-capable classes"; the latter are referred to as "transient classes."

The C++ to ODBMS language binding approach described by this standard is based on the smart pointer or "Ref-based" approach. For each persistence-capable class T, an ancillary class d_Ref<T> is defined. Instances of persistence-capable classes are then referenced using parameterized references, e.g.,

```
(1) d_Ref<Professor>    profP;
(2) d_Ref<Department>   deptRef;
(3) profP->grant_tenure();
(4) deptRef = profP->dept;
```

Statement (1) declares the object profP as an instance of the type d_Ref<Professor>. Statement (2) declares deptRef as an instance of the type d_Ref<Department>. Statement (3) invokes the grant_tenure operation defined on class Professor, on the instance of that class referred to by profP. Statement (4) assigns the value of the dept attribute of the professor referenced by profP to the variable deptRef.

Instances of persistence-capable classes may contain embedded members of C++ built-in types, user-defined classes, or pointers to transient data. Applications may refer to such embedded members using C++ pointers (*) or references (&) only during the execution of a transaction.

In this chapter we use the following terms to describe the places where the standard is formally considered undefined or allows for an implementor of one of the bindings to make implementation-specific decisions with respect to implementing the standard. The terms are

> *Undefined:* The behavior is unspecified by the standard. Implementations have complete freedom (can do anything or nothing), and the behavior need not be documented by the implementor or vendor.
>
> *Implementation-defined:* The behavior is specified by each implementor/vendor. The implementor/vendor is allowed to make implementation-specific decisions about the behavior. However, the behavior must be well defined and fully documented and published as part of the vendor's implementation of the standard.

Figure 5-1 shows the hierarchy of languages involved, as well as the preprocess, compile, and link steps that generate an executable application.

5.1.3 Mapping the ODMG Object Model into C++

Although C++ provides a powerful data model that is close to the one presented in Chapter 2, it is worth trying to explain more precisely how concepts introduced in Chapter 2 map into concrete C++ constructs.

5.1.3.1 Object and Literal

An ODMG object type maps into a C++ class. Depending on how a C++ class is instantiated, the result can be an ODMG object or an ODMG literal. A C++ object embedded as a member within an enclosing class is treated as an ODMG literal. This is explained by the fact that a block of memory is inserted into the enclosing object and belongs entirely to it. For instance, one cannot copy the enclosing object without getting a copy of the embedded one at the same time. In this sense the embedded object cannot be considered as having an identity, since it acts as a literal.

5.1.3.2 Structure

The Object Model notion of a *structure* maps into the C++ construct *struct* or *class* embedded in a class.

5.1.3.3 Implementation

C++ has implicit the notion of dividing a class definition into two parts: its interface (public part) and its implementation (protected and private members and function definitions). However, in C++ only one implementation is possible for a given class.

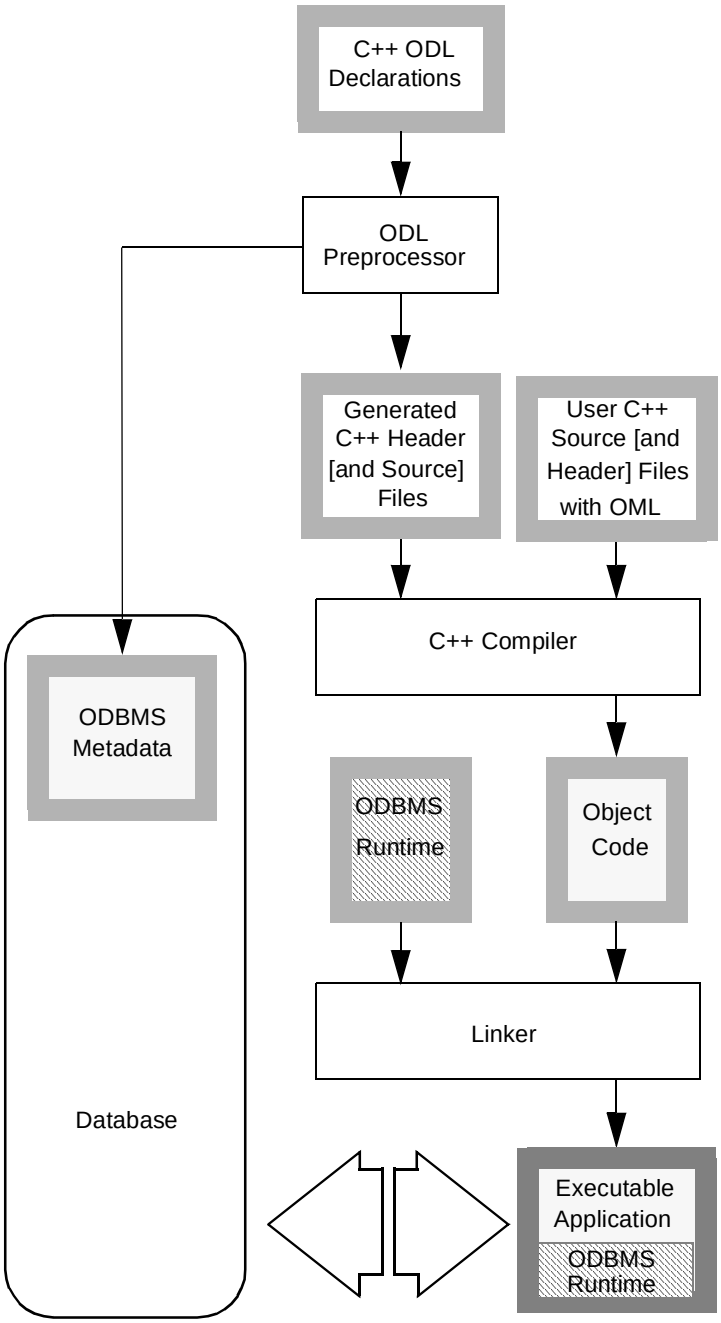


Figure 5-1. Language Hierarchy

5.1.3.4 Collection Classes

The ODMG Object Model includes collection type generators, collection types, and collection instances. Collection type generators are represented as *template classes* in C++. Collection types are represented as collection classes, and collection instances are represented as instances of these collection classes. To illustrate these three categories:

```
template<class T> class d_Set : public d_Collection<T> { ... };
class Ship { ... };
d_Set<d_Ref<Ship> > Cunard_Line;
```

d_Set<T> is a collection template class. d_Set<d_Ref<Ship> > is a collection class. And Cunard_Line is a particular collection, an instance of the class d_Set<d_Ref<Ship> >.

The subtype/supertype hierarchy of collection types defined in the ODMG Object Model is directly carried over into C++. The type d_Collection<T> is an abstract class in C++ with no direct instances. It is instantiable only through its derived classes. The only differences between the collection classes in the C++ binding and their counterparts in the Object Model are the following:

- Named operations in the Object Model are mapped to C++ function members.
- For some operations, the C++ binding includes both the named function and an overloaded infix operation, e.g., d_Set::union_with also has the form operator+=. The statements s1.union_with(s2) and s1 += s2 are functionally equivalent.
- Operations that return a boolean in the Object Model are modeled as function members that return a d_Boolean in the C++ binding.
- The create and delete operations defined in the Object Model have been replaced with C++ constructors and destructors.

5.1.3.5 Array

C++ provides a syntax for creating and accessing a contiguous and indexable sequence of objects. This has been chosen to map partially to the ODMG Array collection. To complement it, a d_Varray C++ class is also provided, which implements an array whose upper bound may vary.

5.1.3.6 Relationship

Relationships are not directly supported by C++. Instead, they are supported in ODMG by including instances of specific template classes that provide the maintenance of the relationship.

The relation itself is implemented as a reference (one-to-one relation) or as a collection (one-to-many relation) embedded in the object.

5.1.3.7 Extents

The class d_Extent<T> provides an interface to the extent for a persistence-capable class T in the C++ binding.

5.1.3.8 Keys

Key declarations are not supported by C++.

5.1.3.9 Names

The previous releases only allowed one name per object in the C++ binding. Now an object can have multiple names. The bind operation in the Object Model is implemented in C++ with the set_object_name and rename_object methods to maintain backward compatibility with previous releases of the C++ binding.

5.1.4 Use of C++ Language Features

5.1.4.1 Prefix

The global names in the ODMG interface will have a prefix of d_. The intention is to avoid name collisions with other names in the global namespace. The ODMG will keep the prefix even after C++ namespaces are generally available.

5.1.4.2 Namespaces

The namespace feature added to C++ did not have generally available implementations at the time this specification was written. In the future the ODMG plans to utilize namespaces and intends to use odmg as its namespace name.

5.1.4.3 Exception Handling

When error conditions are detected, an instance of class d_Error is thrown using the standard C++ exception mechanism. Class d_Error is derived from the class exception defined in the C++ standard.

5.1.4.4 Preprocessor Identifier

A preprocessor identifier, __ODMG_93__, is defined for conditional compilation. With ODMG 2.0 the following symbol:

```
#define __ODMG__ 20
```

is defined. The value of this symbol indicates the specific ODMG release, for example, 20 (release 2.0), 21 (release 2.1), or 30 (release 3.0).

5.1.4.5 Implementation Extensions

Implementations must provide the full function signatures for all the interface methods specified in the chapter and may provide variants on these methods, with additional

parameters. Each additional parameter must have a default value. This allows applications that do not use the additional parameters to be portable.

5.2 C++ ODL

This section defines the C++ Object Definition Language. C++ ODL provides a description of the database schema as a set of object classes—including their attributes, relationships, and operations—in a syntactic style that is consistent with that of the declarative portion of a C++ program. Instances of these classes can be manipulated through the C++ OML.

Following is an example declaring type Professor:

```
extern const char _professors[ ];
extern const char _advisor [ ];

class Professor : public d_Object {
public:
// properties:
      d_UShort                                age;
      d_UShort                                id_number;
      d_String                                office_number;
      d_String                                name;
      d_Rel_Ref<Department, _professors>      dept;
      d_Rel_Set<Student, _advisor>            advisees;
// operations:
      void                                    grant_tenure();
      void                                    assign_course(Course &);
private:
                  ...
};

const char _professors [ ] = "professors";
const char _advisor [ ] = "advisor";
```

The syntax for a C++ ODL class declaration is identical to a C++ class declaration. Attribute declarations map to a restricted set of C++ data member declarations. The variables _professors and _advisor are used for establishing an association between the two ends of a relationship.

Static data members of classes are not contained within each individual instance but are of static storage class. Thus static data members are not stored in the database, but are supported for persistence-capable classes. Supertypes are specified using the stan-

dard C++ syntax within the class header, e.g., class Professor : public Employee. Though this specification may use public members for ease and brevity, private and protected members are supported.

5.2.1 Attribute Declarations

Attribute declarations are syntactically identical to data member declarations within C++. Because notions of attributes as objects are not yet defined and included in this standard, attributes and data members are not and cannot be syntactically distinguished. In this standard, an attribute cannot have properties (e.g., unit of measure) and there is no way to specialize the get_value and set_value operations defined on the type (e.g., to raise an event when a value is changed).

Standard C++ syntax and semantics for class definitions are supported. However, compliant implementations need not support the following data types within persistent classes:

- unions
- bit fields
- references (&)

as members. Unions and bit fields pose problems when supporting heterogeneous environments. The semantics of references is that they are initialized once at creation; all subsequent operations are directed to the referenced object. References within persistent objects cannot be reinitialized when brought from the database into memory and their initialization value would, in general, not be valid across process boundaries. A set of special classes is defined within the ODMG specification to contain references to persistent objects.

In addition to all primitive data types, except those noted above, structures and class objects can be members. There are several structured literal types that are provided. These include:

- d_String
- d_Interval
- d_Date
- d_Time
- d_Timestamp

Examples:

```
struct University_Address {
    d_UShort    PO_box;
    d_String    university;
    d_String    city;
    d_String    state;
    d_String    zip_code;
};
```

```
class Student : public d_Object {
public:
    d_String              name;
    d_Date                birth_date;
    Phone_Number          dorm_phone;
    University_Address    address;
    d_List<d_String>      favorite_friends;
};
```

The attribute name takes a d_String as its value. The attribute dorm_phone takes a user-defined type Phone_Number as its value. The attribute address takes a structure. The attribute favorite_friends takes a d_List of d_String as its value. The following sections contain descriptions of the provided literal types.

5.2.1.1 Fixed-Length Types

In addition to the C++ built-in data types, such as the signed, unsigned, and floating point numeric data types, the following fixed-length types will be supported for use in defining attributes of persistence capable classes.

Type Name	Range	Description
d_Short	16 bit	signed integer
d_Long	32 bit	signed integer
d_UShort	16 bit	unsigned integer
d_ULong	32 bit	unsigned integer
d_Float	32 bit	IEEE Std 754-1985 single-precision floating point
d_Double	64 bit	IEEE Std 754-1985 double-precision floating point
d_Char	8 bit	ASCII
d_Octet	8 bit	no interpretation
d_Boolean	d_True or d_False	defines d_True (nonzero value) and d_False (zero value)

Unlike the C++ built-in types, these types have the same range and interpretation on all platforms and environments. Use of these types is recommended when developing applications targeted for heterogeneous environments. Note that like all other global names described in this chapter, these types will be defined within the ODMG namespace when that feature becomes available.

Any ODMG implementation that allows access to a database from applications that have been constructed with different assumptions about the range or interpretation of

the C++ built-in types may require the use of the fixed-length data types listed above when defining attributes of persistent objects. The behavior of the database system in such a heterogeneous environment when the C++ built-in types are used for persistent data attributes is undefined.

ODMG implementations will allow but not require the use of the fixed-length data types when used in homogeneous environments.

For any given C++ language environment or platform, these fixed-length data types may be defined as identical to a built-in C++ data type that conforms to the range and interpretation requirements. Since a given C++ built-in data type may meet the requirements in some environments but not in others, portable application code should not assume any correspondence or lack of correspondence between the fixed-length data types and similar C++ built-in data types. In particular, function overloads should not be disambiguated solely on the difference between a fixed-length data type and a closely corresponding C++ built-in data type. Also, different implementations of a virtual function should use signatures that correspond exactly to the declaration in the base class with respect to use of fixed-length data types versus C++ built-in data types.

5.2.1.2 d_String

The following class defines a literal type to be used for string attributes. It is intended that this class is used strictly for storing strings in the database, as opposed to being a general string class with all the functionality of a string class normally used for transient strings in an application.

Initialization, assignment, copying, and conversion to and from C++ character strings are supported. The comparison operators are defined on d_String to compare with either another d_String or a C++ character string. One can also access an element in the d_String via an index and also determine the length of the d_String.

Definition:

```
class d_String {
public:
                        d_String();
                        d_String(const d_String &);
                        d_String(const char *);
                        ~d_String();
    d_String &          operator=(const d_String &);
    d_String &          operator=(const char *);
                        operator const char *() const;
    char &              operator[](unsigned long index);
    unsigned long       length() const;
```

```
friend  d_Boolean       operator==(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator==(const d_String &sL, const char *pR);
friend  d_Boolean       operator==(const char *pL, const d_String &sR);
friend  d_Boolean       operator!= (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator!= (const d_String &sL, const char *pR);
friend  d_Boolean       operator!= (const char *pL, const d_String &sR);
friend  d_Boolean       operator<   (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator<   (const d_String &sL, const char *pR);
friend  d_Boolean       operator<   (const char *pL, const d_String &sR);
friend  d_Boolean       operator<=(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator<=(const d_String &sL, const char *pR);
friend  d_Boolean       operator<=(const char *pL, const d_String &sR);
friend  d_Boolean       operator>   (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator>   (const d_String &sL, const char *pR);
friend  d_Boolean       operator>   (const char *pL, const d_String &sR);
friend  d_Boolean       operator>=(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator>=(const d_String &sL, const char *pR);
friend  d_Boolean       operator>=(const char *pL, const d_String &sR);
};
```

Class d_String is responsible for freeing the string that gets returned by operator const char *.

5.2.1.3 d_Interval

The d_Interval class is used to represent a duration of time. It is also used to perform arithmetic operations on the d_Date, d_Time, and d_Timestamp classes. This class corresponds to the day-time interval as defined in the SQL standard.

Initialization, assignment, arithmetic, and comparison functions are defined on the class, as well as member functions to access the time components of its current value.

The d_Interval class accepts nonnormalized input, but normalizes the time components when accessed. For example, the constructor would accept 28 hours as input, but then calling the day function would return a value of 1 and the hour function would return a value of 4. Arithmetic would work in a similar manner.

Definition:

```
class d_Interval {
public:
                        d_Interval(int day = 0, int hour = 0,int min = 0, float sec = 0.0);
                        d_Interval(const d_Interval &);
    d_Interval &        operator=(const d_Interval &);
    int                 day() const;
```

```
    int             hour() const;
    int             minute() const;
    float           second() const;
    d_Boolean       is_zero() const;
    d_Interval &    operator+=(const d_Interval &);
    d_Interval &    operator–=(const d_Interval &);
    d_Interval &    operator*=(int);
    d_Interval &    operator/=(int);
    d_Interval      operator–() const;
friend  d_Interval  operator+(const d_Interval &L, const d_Interval &R);
friend  d_Interval  operator–(const d_Interval &L, const d_Interval &R);
friend  d_Interval  operator*(const d_Interval &L, int R);
friend  d_Interval  operator*(int L, const d_Interval &R);
friend  d_Interval  operator/ (const d_Interval &L, int R);
friend  d_Boolean   operator==(const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator!= (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator<  (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator<=(const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator>  (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator>=(const d_Interval &L, const d_Interval &R);
};
```

5.2.1.4 d_Date

The d_Date class stores a representation of a date consisting of a year, month, and day. It also provides enumerations to denote weekdays and months.

Initialization, assignment, arithmetic, and comparison functions are provided. Implementations may have additional functions available to support converting to and from the type used by the operating system to represent a date. Functions are provided to access the components of a date. There are also functions to determine the number of days in a month, etc. The static function current returns the current date. The next and previous functions advance the date to the next specified weekday.

Definition:

```
class d_Date {
public:
    enum Weekday {
        Sunday = 0,     Monday = 1,     Tuesday = 2,     Wednesday = 3,
        Thursday = 4,   Friday = 5,     Saturday = 6
    };
    enum Month {
```

```
        January = 1,   February = 2,   March = 3,   April = 4,   May = 5,   June = 6,
        July = 7,    August = 8,   September = 9,   October = 10,   November = 11,
        December = 12
    };
                        d_Date();           // sets to current date
                        d_Date(unsigned short year, unsigned short day_of_year);
                        d_Date(unsigned short year, unsigned short month,
                                unsigned short day);
                        d_Date(const d_Date &);
                        d_Date(const d_Timestamp &);
    d_Date &            operator=(const d_Date &);
    d_Date &            operator=(const d_Timestamp &);
    unsigned short      year() const;
    unsigned short      month() const;
    unsigned short      day() const;
    unsigned short      day_of_year() const;
    Weekday             day_of_week() const;
    Month               month_of_year() const;
    d_Boolean           is_leap_year() const;
static  d_Boolean       is_leap_year(unsigned short year);
static  d_Date          current();
    d_Date &            next(Weekday);
    d_Date &            previous(Weekday);
    d_Date &            operator+=(const d_Interval &);
    d_Date &            operator+=(int ndays);
    d_Date &            operator++();           // prefix  ++d
    d_Date              operator++(int);        // postfix  d++
    d_Date &            operator-=(const d_Interval &);
    d_Date &            operator-=(int ndays);
    d_Date &            operator--();           // prefix --d
    d_Date              operator--(int);        // postfix  d--
friend  d_Date          operator+(const d_Date &L, const d_Interval &R);
friend  d_Date          operator+(const d_Interval &L, const d_Date &R);
friend  d_Interval      operator-(const d_Date &L, const d_Date &R);
friend  d_Date          operator-(const d_Date &L, const d_Interval &R);
friend  d_Boolean       operator==(const d_Date &L, const d_Date &R);
friend  d_Boolean       operator!= (const d_Date &L, const d_Date &R);
friend  d_Boolean       operator<  (const d_Date &L, const d_Date &R);
friend  d_Boolean       operator<=(const d_Date &L, const d_Date &R);
friend  d_Boolean       operator>  (const d_Date &L, const d_Date &R);
friend  d_Boolean       operator>=(const d_Date &L, const d_Date &R);
```

```
    d_Boolean       is_between(const d_Date &, const d_Date &) const;
friend d_Boolean    overlaps(const d_Date &psL, const d_Date &peL,
                             const d_Date &psR, const d_Date &peR);
friend d_Boolean    overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                             const d_Date &sR, const d_Date &eR);
friend d_Boolean    overlaps(const d_Date &sL, const d_Date &eL,
                             const d_Timestamp &sR, const d_Timestamp &eR);
static int          days_in_year(unsigned short year);
    int             days_in_year() const;
static int          days_in_month(unsigned short yr, unsigned short month);
    int             days_in_month() const;
static d_Boolean    is_valid_date(unsigned short year, unsigned short month,
                                  unsigned short day);
};
```

If an attempt is made to set a d_Date object to an invalid value, a d_Error exception object of kind d_Error_DateInvalid is thrown and the value of the d_Date object is undefined.

The functions next, previous, operator+=, and operator–= alter the object and return a reference to the current object. The post increment and decrement operators return a new object by value.

The overlaps functions take two periods (start and end), each period denoted by a start and end time, and determines whether the two time periods overlap. The is_between function determines whether the d_Date value is within a given period.

5.2.1.5 d_Time

The d_Time class is used to denote a specific time, which is internally stored in Greenwich Mean Time (GMT). Initialization, assignment, arithmetic, and comparison operators are defined. There are also functions to access each of the components of a time value. Implementations may have additional functions available to support converting to and from the type used by the operating system to represent a time.

The enumeration Time_Zone is made available to denote a specific time zone. Time zones are numbered according to the number of hours that must be added or subtracted from local time to get the time in Greenwich, England (GMT). Thus the value of GMT is 0. A Time_Zone name of GMT6 indicates a time of 6 hours greater than GMT, and thus 6 must be subtracted from it to get GMT. Conversely, GMT_8 means that the time is 8 hours earlier than GMT (read the underscore as a minus). A default time zone value is maintained and is initially set to the local time zone. It is possible to change the default time zone value as well as reset it to the local value.

Definition:

```
class d_Time {
public:
    enum Time_Zone {
        GMT   = 0,   GMT12 = 12,     GMT_12 = –12,
        GMT1 = 1,    GMT_1 = –1,     GMT2 = 2,      GMT_2 = –2,
        GMT3 = 3,    GMT_3 =  –3,    GMT4 = 4,      GMT_4 = –4,
        GMT5 = 5,    GMT_5 = –5,     GMT6 = 6,      GMT_6 = –6,
        GMT7 = 7,    GMT_7 = –7,     GMT8 = 8,      GMT_8 = –8,
        GMT9 = 9,    GMT_9 = –9,     GMT10 = 10,    GMT_10 = –10,
        GMT11 = 11, GMT_11 = –11,
        USeastern = –5,   UScentral = –6, USmountain = –7, USpacific = –8
    };
static   void             set_default_Time_Zone(Time_Zone);
static   void             set_default_Time_Zone_to_local();
                          d_Time();
                          d_Time(unsigned short hour,
                                  unsigned short minute, float sec = 0.0f);
                          d_Time(unsigned short hour, unsigned short minute,
                                  float sec, short tzhour, short tzminute);
                          d_Time(const d_Time &);
                          d_Time(const d_Timestamp &);
    d_Time &              operator=(const d_Time &);
    d_Time &              operator=(const d_Timestamp &);
    unsigned short        hour() const;
    unsigned short        minute() const;
    Time_Zone             time_zone() const;
    float                 second() const;
    short                 tz_hour() const;
    short                 tz_minute() const;
static   d_Time           current();
    d_Time &              operator+=(const d_Interval &);
    d_Time &              operator–=(const d_Interval &);
friend  d_Time            operator+(const d_Time &L, const d_Interval &R);
friend  d_Time            operator+(const d_Interval &L, const d_Time &R);
friend  d_Interval        operator–(const d_Time &L, const d_Time &R);
friend  d_Time            operator–(const d_Time &L, const d_Interval &R);
friend  d_Boolean         operator==(const d_Time &L, const d_Time &R);
friend  d_Boolean         operator!= (const d_Time &L, const d_Time &R);
friend  d_Boolean         operator<  (const d_Time &L, const d_Time &R);
```

```
friend  d_Boolean       operator<=(const d_Time &L, const d_Time &R);
friend  d_Boolean       operator>  (const d_Time &L, const d_Time &R);
friend  d_Boolean       operator>=(const d_Time &L, const d_Time &R);
friend  d_Boolean       overlaps(const d_Time &psL, const d_Time &peL,
                                 const d_Time &psR, const d_Time &peR);
friend  d_Boolean       overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                                 const d_Time &sR, const d_Time &eR);
friend  d_Boolean       overlaps(const d_Time &sL, const d_Time &eL,
                                 const d_Timestamp &sR, const d_Timestamp &eR);
};
```

All arithmetic on d_Time is done on a modulo 24-hour basis. If an attempt is made to set a d_Time object to an invalid value, a d_Error exception object of kind d_Error_TimeInvalid is thrown and the value of the d_Time object is undefined.

The default d_Time constructor initializes the object to the current time. The overlaps functions take two periods, each denoted by a start and end time, and determines whether the two time periods overlap.

5.2.1.6 d_Timestamp

A d_Timestamp consists of both a date and time.

Definition:

```
class d_Timestamp {
public:
                        d_Timestamp();      // sets to the current date/time
                        d_Timestamp(unsigned short year, unsigned short month=1,
                                unsigned short day = 1, unsigned short hour = 0,
                                unsigned short minute = 0, float sec = 0.0);
                        d_Timestamp(const d_Date &);
                        d_Timestamp(const d_Date &, const d_Time &);
                        d_Timestamp(const d_Timestamp &);
    d_Timestamp &       operator=(const d_Timestamp &);
    d_Timestamp &       operator=(const d_Date &);
    const d_Date &      date() const;
    const d_Time &      time() const;
    unsigned short      year() const;
    unsigned short      month() const;
    unsigned short      day() const;
    unsigned short      hour() const;
    unsigned short      minute() const;
    float               second() const;
    short               tz_hour() const;
    short               tz_minute() const;
```

```
static   d_Timestamp   current();
      d_Timestamp &    operator+=(const d_Interval &);
      d_Timestamp &    operator-=(const d_Interval &);
friend  d_Timestamp    operator+(const d_Timestamp &L, const d_Interval &R);
friend  d_Timestamp    operator+(const d_Interval &L, const d_Timestamp &R);
friend  d_Timestamp    operator-(const d_Timestamp &L, const d_Interval &R);
friend  d_Interval     operator-(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator==(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator!= (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator<  (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator<=(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator>  (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      operator>=(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean      overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                                const d_Timestamp &sR, const d_Timestamp &eR);
friend  d_Boolean      overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                                const d_Date &sR, const d_Date &eR);
friend  d_Boolean      overlaps(const d_Date &sL, const d_Date &eL,
                                const d_Timestamp &sR, const d_Timestamp &eR);
friend  d_Boolean      overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                                const d_Time &sR, const d_Time &eR);
friend  d_Boolean      overlaps(const d_Time &sL, const d_Time &eL,
                                const d_Timestamp &sR, const d_Timestamp &eR);
};
```

If an attempt is made to set the value of a d_Timestamp object to an invalid value, a d_Error exception object of kind d_Error_TimestampInvalid is thrown and the value of the d_Timestamp object is undefined.

5.2.2 Relationship Traversal Path Declarations

Relationships do not have syntactically separate definitions. Instead, the *traversal paths* used to cross relationships are defined within the bodies of the definitions of each of the two object types that serve a role in the relationship. For example, if there is a one-to-many relationship between professors and the students they have as advisees, then the traversal path advisees is defined within the type definition of the object type Professor**,** and the traversal path advisor is defined within the type definition of the object type Student.

A relationship traversal path declaration is similar to an attribute declaration, but with the following differences. Each end of a relationship has a relationship traversal path. A traversal path declaration is an attribute declaration and must be of type

- d_Rel_Ref<T, const char *> (which has the interface of d_Ref<T>)
- d_Rel_Set<T, const char *> (which has the interface of d_Set<d_Ref<T> >)
- d_Rel_List<T, const char *> (which has the interface of d_List<d_Ref<T> >)

for some persistent class T. The second template argument should be a variable that contains the name of the attribute in the other class, which serves as the inverse role in the relationship. Both classes in a relationship must have a member of one of these types, and the members of the two classes must refer to each other. If the second template argument has a name that does not correspond to a data member in the referenced class, a d_Error exception object of kind d_Error_MemberNotFound is thrown. If the second template argument does refer to a data member, but the data member is the wrong type, i.e., it is not of type d_Rel_Ref, d_Rel_Set, or d_Rel_List, a d_Error exception object of kind d_Error_MemberIsOfInvalidType is thrown.

Studying the relationships in the examples below will make this clear.

Examples:

```
extern const char _dept [ ],       _professors [ ] ;
extern const char _advisor [ ],    _advisees [ ] ;
extern const char _classes [ ],    _enrolled [ ] ;

class Department : public d_Object {
public:
    d_Rel_Set<Professor, _dept>             professors;
};
class Professor : public d_Object {
public:
    d_Rel_Ref<Department, _professors>      dept;
    d_Rel_Set<Student, _advisor>            advisees;
};
class Student : public d_Object {
public:
    d_Rel_Ref<Professor, _advisees>         advisor;
    d_Rel_Set<Course, _enrolled>            classes;
};
class Course : public d_Object {
public:
    d_Rel_Set<Student, _classes>            students_enrolled;
};
const char _dept [ ] = "dept";
const char _professors [ ] = "professors";
```

```
const char _advisor [ ] = "advisor";
const char _advisees [ ] = "advisees";
const char _classes [ ] = "classes" ;
const char _enrolled [ ] = "students_enrolled";
```

The second template parameter is based on the address of the variable, not on the string contents. Thus a different variable is required for each role, even if the member name happens to be the same. The string contents must match the name of the member in the other class involved in the relationship.

The referential integrity of bidirectional relationships is automatically maintained. If a relationship exists between two objects and one of the objects gets deleted, the relationship is considered to no longer exist and the inverse traversal path will be altered to remove the relationship.

5.2.3 Operation Declarations

Operation declarations in C++ are syntactically identical to *function member* declarations. For example, see grant_tenure and assign_course defined for class Professor in Section 5.2.

5.3 C++ OML

This section describes the C++ binding for the OML. A guiding principle in the design of C++ OML is that the syntax used to create, delete, identify, reference, get/set property values, and invoke operations on a persistent object should be, so far as possible, no different than that used for objects of shorter lifetimes. A single expression may freely intermix references to persistent and transient objects.

While it is our long-term goal that nothing can be done with persistent objects that cannot also be done with transient objects, this standard treats persistent and transient objects slightly differently. Queries and transaction consistency apply only to persistent objects.

5.3.1 Object Creation, Deletion, Modification, and References

Objects can be created, deleted, and modified. Objects are created in C++ OML using the new operator, which is overloaded to accept additional arguments specifying the lifetime of the object. An optional storage pragma allows the programmer to specify how the newly allocated object is to be clustered with respect to other objects.

The static member variable d_Database::transient_memory is defined in order to allow libraries that create objects to be used uniformly to create objects of any lifetime. This variable may be used as the value of the database argument to operator new to create objects of transient lifetime.

```
static d_Database * const d_Database::transient_memory;
```

The formal ODMG forms of the C++ new operator are

```
(1) void * operator new(size_t size);
(2) void * operator new(size_t size, const d_Ref_Any &clustering,
                        const char* typename);
(3) void * operator new(size_t size, d_Database *database,
                        const char* typename);
```

These operators have d_Object scope. (1) is used for creation of transient objects derived from d_Object. (2) and (3) create persistent objects. In (2) the user specifies that the newly created object should be placed "near" the existing clustering object. The exact interpretation of "near" is implementation-defined. An example interpretation would be "on the same page if possible." In (3) the user specifies that the newly created object should be placed in the specified database, but no further clustering is specified.

The size argument, which appears as the first argument in each signature, is the size of the representation of an object. It is determined by the compiler as a function of the class of which the new object is an instance, not passed as an explicit argument by a programmer writing in the language.

If the database does not have the schema information about a class when new is called, a d_Error exception object of kind d_Error_DatabaseClassUndefined is thrown.

Examples:

```
    d_Database *yourDB, *myDB;  // assume these get initialized properly
(1) d_Ref<Schedule> temp_sched1 = new Schedule;
(2) d_Ref<Professor> prof2 = new(yourDB,"Professor") Professor;
(3) d_Ref<Student> student1 = new(myDB, "Student") Student;
(4) d_Ref<Student> student2 = new(student1, "Student") Student;
(5) d_Ref<Student> temp_student =
                        new(d_Database::transient_memory, "Student") Student;
```

Statement (1) creates a transient object temp_sched1. Statements (2)–(4) create persistent objects. Statement (2) creates a new instance of class Professor in the database yourDB. Statement (3) creates a new instance of class Student in the database myDB. Statement (4) does the same thing, except that it specifies that the new object, student2, should be placed close to student1. Statement (5) creates a transient object temp_student.

5.3.1.1 Object Deletion

Objects, once created, can be deleted in C++ OML using the d_Ref::delete_object member function. Using the delete operator on a pointer to a persistent object will also delete the object, as in standard C++ practice. Deleting an object is permanent, subject to transaction commit. The object is removed from memory and, if it is a persistent object, from the database. The d_Ref instance or pointer still exists in memory, but its

reference value is undefined. An attempt to access the deleted object is implementation defined.

Example:

```
d_Ref<anyType> obj_ref;
 ...   // set obj_ref to refer to a persistent object
obj_ref.delete_object();
```

C++ requires the operand of delete to be a pointer, so the member function delete_object was defined to delete an object with just a d_Ref<T> reference to it.

5.3.1.2 Object Modification

The state of an object is modified by updating its properties or by invoking operations on it. Updates to persistent objects are made visible to other users of the database when the transaction containing the modifications commits.

Persistent objects that will be modified must communicate to the runtime ODBMS process the fact that their states will change. The ODBMS will then update the database with these new states at transaction commit time. Object change is communicated by invoking the d_Object::mark_modified member function, which is defined in Section 5.3.4 and is used as follows:

```
obj_ref–>mark_modified();
```

The mark_modified function call is included in the constructor and destructor methods for persistence-capable classes, i.e., within class d_Object. The developer should include the call in any other methods that modify persistent objects, before the object is actually modified.

As a convenience, the programmer may omit calls to mark_modified on objects where classes have been compiled using an optional C++ OML preprocessor switch; the system will automatically detect when the objects are modified. In the default case, mark_modified calls are required, because in some ODMG implementations performance will be better when the programmer explicitly calls mark_modified. However, each time a persistent object is modified by a member update function provided explicitly by the ODMG classes, the mark_modified call is not necessary since it is done automatically.

5.3.1.3 Object References

Objects, whether persistent or not, may refer to other objects via object references. In C++ OML object references are instances of the template class d_Ref<T> (see Section 5.3.5). All accesses to persistent objects are made via methods defined on classes d_Ref, d_Object, and d_Database. The dereference operator –> is used to access

members of the persistent object "addressed" by a given object reference. How an object reference is converted to a C++ pointer to the object is implementation-defined.

A dereference operation on an object reference always guarantees that the object referred to is returned or a d_Error exception object of kind d_Error_RefInvalid is thrown. The behavior of a reference is as follows. If an object reference refers to a persistent object that exists but is not in memory when a dereference is performed, it will be retrieved automatically from disk, mapped into memory, and returned as the result of the dereference. If the referenced object does not exist, a d_Error exception object of kind d_Error_RefInvalid is thrown. References to transient objects work exactly the same (at least on the surface) as references to persistent objects.

Any object reference may be set to a null reference or *cleared* to indicate the reference does not refer to an object.

The rules for when an object of one lifetime may refer to an object of another lifetime are a straightforward extension of the C++ rules for its two forms of transient objects—procedure coterminus and process coterminus. An object can always refer to another object of longer lifetime. An object can only refer to an object of shorter lifetime as long as the shorter-lived object exists.

A persistent object is retrieved from disk upon activation. It is the application's responsibility to initialize the values of any of that object's pointers to transient objects. When a persistent object is committed, the ODBMS sets its embedded d_Refs to transient objects to the null value.

5.3.1.4 Object Names

A database application generally will begin processing by accessing one or more critical objects and proceeding from there. These objects are in some sense "root" objects, in that they lead to interconnected webs of other objects. The ability to name an object and retrieve it later by that name facilitates this start-up capability. Named objects are also convenient in many other situations.

There is a single, flat name scope per database; thus all names in a particular database are unique. A name is not explicitly defined as an attribute of an object. The operations for manipulating names are defined in the d_Database class in Section 5.3.8.

5.3.2 Properties

5.3.2.1 Attributes

C++ OML uses standard C++ for accessing attributes. For example, assume prof has been initialized to reference a professor and we wish to modify its id_number:

```
prof–>id_number = next_id;
cout << prof–>id_number;
```

Modifying an attribute's value is considered a modification to the enclosing object instance. One must call mark_modified for the object before it is modified.

The C++ binding allows persistence-capable classes to embed instances of C++ classes, including other persistence-capable classes. However, embedded objects are not considered "independent objects" and have no object identity of their own. Users are not permitted to get a d_Ref to an embedded object. Just as with any attribute, modifying an embedded object is considered a modification to the enclosing object instance, and mark_modified for the enclosing object must be called before the embedded object is modified.

5.3.2.2 Relationships

The ODL specifies which relationships exist between object classes. Creating, traversing, and breaking relationships between instances are defined in the C++ OML. Both to-one and to-many traversal paths are supported by the OML. The integrity of relationships is maintained by the ODBMS.

The following diagrams will show graphically the effect of adding, modifying, and deleting relationships among classes. Each diagram is given a name to reflect the cardinality and resulting effect on the relationship. The name will begin with 1-1, 1-m, or m-m to denote the cardinality and will end in either N (no relationship), A (add a relationship), or M (modify a relationship). When a relationship is deleted, this will result in a state of having no relationship (N). A solid line is drawn to denote the explicit operation performed by the program, and a dashed line shows the side effect operation performed automatically by the ODBMS to maintain referential integrity.

The following template class allows one to specify a to-one relationship to a class T.

```
template <class T, const char *Member> class d_Rel_Ref : public d_Ref<T> {  };
```

The template d_Rel_Ref<T,M> supports the same interface as d_Ref<T>. Implementations will redefine some functions to provide support for referential integrity.

The application programmer must introduce two const char * variables, one used at each end of the relationship to refer to the other end of the relationship, thus establishing the association of the two ends of the relationship. The variables must be initialized with the name of the attribute at the other end of the relationship.

Assume the following 1-1 relationship exists between class A and class B:

```
extern const char _ra [ ], _rb [ ] ;
class A {
      d_Rel_Ref<B, _ra>     rb;
};
```

```
class B {
      d_Rel_Ref<A, _rb>     ra;
};
const char _ra [ ] = "ra";
const char _rb [ ] = "rb";
```

Note that class A and B could be the same class, as well. In each of the diagrams below, there will be an instance of A called a or aa and an instance of B called b or bb. In the following scenario 1-1N there is no relationship between a and b.

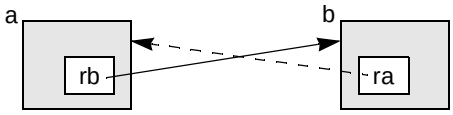
1-1N: No relationship

Then, adding a relationship between a and b via

```
a.rb = &b;
```

results in the following:

1-1A: Add a relationship



The solid arrow indicates the operation specified by the program, and the dashed line shows what operation gets performed automatically by the ODBMS.

Assume now the previous diagram (1-1A) represents the current state of the relationship between a and b. If the program executes the statement:

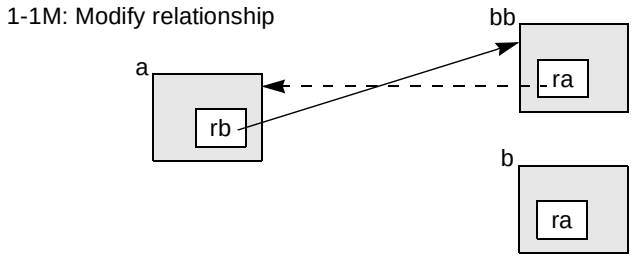
```
a.rb.clear ();
```

the result will be no relationship, as shown in 1-1N.

Assume we have the relationship depicted in 1-1A. If we now execute

```
a.rb = &bb;
```

we obtain the following:



Notice that b.ra no longer refers to A and bb.ra is set automatically to reference a.

Whenever the operand to initialization or assignment represents a null reference, the result will be no relationship as in 1-1N. In the case of assignment, if there had been a relationship, it is removed. If the relationship is currently null (is_null would return true), then doing an assignment would add a relationship, unless the assignment operand was null as well.

If there is currently a relationship with an object, then doing the assignment will modify the relationship as in 1-1M. If the assignment operand is null, then the existing relationship is removed.

When an object involved in a relationship is deleted, all the relationships that the object was involved in will be removed as well.

There are two other cardinalities to consider: one-to-many and many-to-many. With one-to-many and many-to-many relationships, the set of operations allowed are based upon whether the relationship is an unordered set or positional.

The following template class allows one to specify an unordered to-many relationship with a class T:

```
template <class T, const char *M> class d_Rel_Set : public d_Set<d_Ref<T> > { }
```

The template d_Rel_Set<T,M> supports the same interface as d_Set<d_Ref<T> >. Implementations will redefine some functions in order to support referential integrity.

Assuming an unordered one-to-many set relationship between class A and class B:

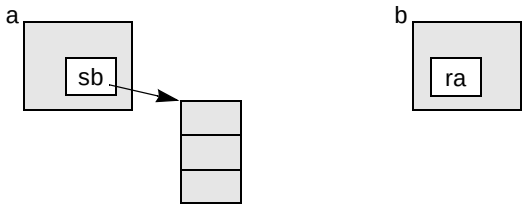
```
extern const char _ra [ ], _sb [ ] ;

class A {
      d_Rel_Set<B, _ra>     sb;
};
class B {
      d_Rel_Ref<A, _sb>     ra;
};
```

```
const char _ra[ ] = "ra";
const char _sb[ ] = "sb";
```

Assume we have the following instances a and b with no relationship.

1-mN: No relationship



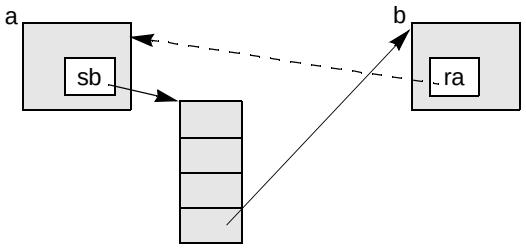
a.sb has 3 elements, but they are referring to instances of B other than b.

Now suppose we add a relationship between a and b by executing the statement

```
a.sb.insert_element (&b);
```

This results in the following:

1-mA: Add a relationship



The b.ra traversal path gets set automatically to reference a. Conversely, if we execute the statement

```
b.ra = &a;
```

an element would have automatically been added to a.sb to refer to b. But only one of the two operations needs to be performed by the program; the ODBMS automatically updates the inverse traversal path.

Given the situation depicted in 1-mA, if we execute either

```
a.sb.remove_element (&b)
```

or

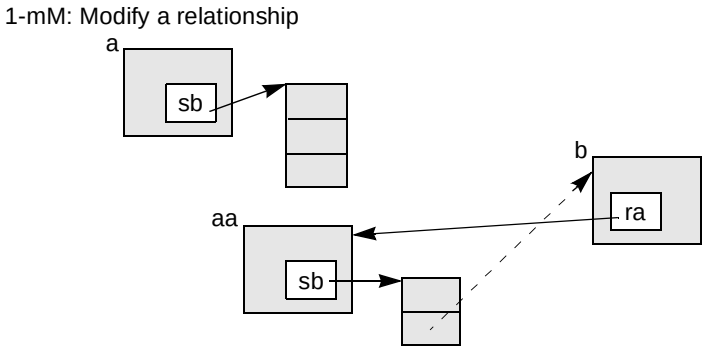
```
b.ra.clear ();
```

the result would be that the relationship between a and b would be deleted and the state of a and b would be as depicted in 1-mN.

Now assume we have the relationship between a and b as shown in 1-mA. If we execute the following statement:

```
b.ra = &aa;
```

this results in the following:



After the statement executes, b.ra refers to aa, and as a side effect, the element within a.sb that had referred to b is removed and an element is added to aa.sb to refer to b.

The d_List class represents a *positional* collection, whereas the d_Set class is an unordered collection. Likewise, the d_Rel_List<T, Member> template is used for representing relationships that are positional in nature.

```
template <class T, const char *M> class d_Rel_List : public d_List<d_Ref<T> > { };
```

The template d_Rel_List<T,M> has the same interface as d_List<d_Ref<T> >.

Assuming a positional to-many relationship between class A and class B:

```
extern const char _ra [ ] , _listB [ ] ;

class A {
      d_Rel_List<B, _ra>          listB;
};
class B {
      d_Rel_Ref<A, _listB>        ra;
};
const char _ra [ ] = "ra";
const char _listB [ ] = "listB";
```

The third relationship cardinality to consider is many-to-many. Suppose we have the following relationship between A and B:

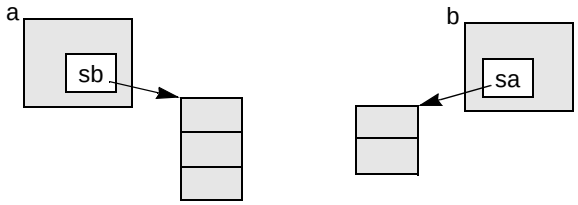
```
extern const char _sa [ ], _sb [ ] ;

class A {
      d_Rel_Set<B, _sa>     sb;
};
```

```
class B {
      d_Rel_Set<A, _sb>     sa;
};
const char _sa [ ] = "sa";
const char _sb [ ] = "sb";
```

Initially, there will be no relationship between instances a and b though a and b have relationships with other instances.

m-mN: No relationship

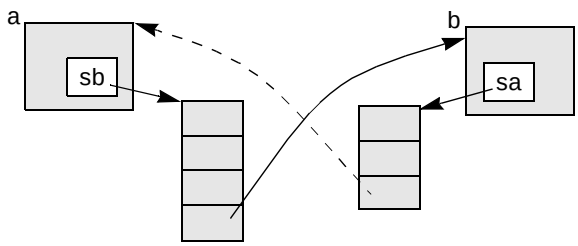


The following statement will add a relationship between a and b:

```
a.sb.insert_element(&b);
```

This will result in the following:

m-mA: Add a relationship



In addition to an element being added to a.sb to reference b, there is an element automatically added to b.sa that references a.

Executing either

```
a.sb.remove_element(&b)
```

or

```
b.sa.remove_element(&a)
```

would result in the relationship being removed between a and b, and the result would be as depicted in m-mN.

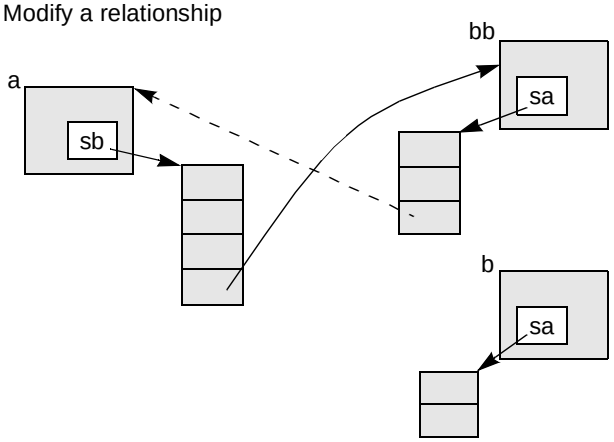
Last, we consider the modification of a many-to-many relationship. Assume the prior state is the situation depicted in m-mA, and assume that sb represents a positional rela-

tionship. The following statement will modify an existing relationship that exists between a and b, changing a to be related to bb.

```
a.sb.replace_element_at( &bb, 3);
```

This results in the following object relationships:

m-mM: Modify a relationship



The result of this operation is that the element in b.sa that referenced a is removed and an element is added to bb.sa to reference a.

The initializations and assignments that have an argument of type d_Rel_Set<T,Member> or d_Set<d_Ref<T> > are much more involved than the simple diagrams above because they involve performing the corresponding operation for every element of the set versus doing it for just one element. The remove_all function removes every member of the relationship, also removing the back-reference for each referenced member. If the assignment operators have an argument that represents an empty set, the assignment will have the same effect as the remove_all function.

Below are some more examples based on the classes used throughout the chapter.

Examples:

```
d_Ref<Professor> p;
d_Ref<Student> Sam;
d_Ref<Department> english_dept;
// initialize p, Sam, and english_dept references
p–>dept = english_dept;   // create 1:1 relationship
p–>dept.clear();              // clear the relationship
p–>advisees.insert_element(Sam); // add Sam to the set of students that are p's
                                 // advisees; same effect as 'Sam–>advisor = p'
p–>advisees.remove_element(Sam); // remove Sam from the set of students that
                                 // are p's advisees, also clears Sam–>advisor
```

5.3.3 Operations

Operations are defined in the OML as they are generally implemented in C++. Operations on transient and persistent objects behave entirely consistently with the operational context defined by standard C++. This includes all overloading, dispatching, function call structure and invocation, member function call structure and invocation, argument passing and resolution, error handling, and compile time rules.

5.3.4 d_Object Class

The class d_Object is introduced and defined as follows:

Definition:

```
class d_Object {
public:
                        d_Object();
                        d_Object(const d_Object &);
virtual                 ~d_Object();
    d_Object &          operator=(const d_Object &);
    void                mark_modified();  // mark the object as modified
    void *              operator new(size_t size);
    void *              operator new(size_t size, const d_Ref_Any &cluster,
                                const char *typename);
    void *              operator new(size_t size, d_Database *database,
                                const char *typename);
    void                operator delete(void *);
virtual  void           d_activate();
virtual  void           d_deactivate();
};
```

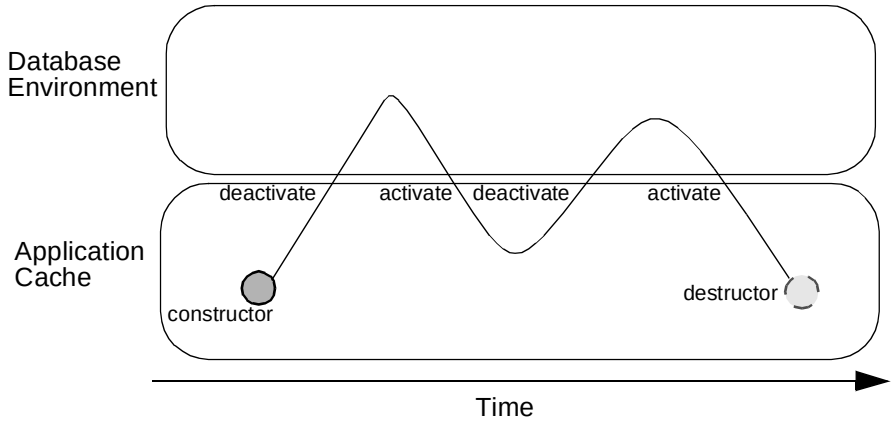
This class is introduced to allow the type definer to specify when a class is capable of having persistent as well as transient instances. Instances of classes derived from d_Object can be either persistent or transient. A class A that is persistence-capable would inherit from class d_Object:

```
class My_Class : public d_Object {...};
```

The delete operator can be used with a pointer to a persistent object to delete the object; the object is removed from both the application cache and the database, which is the same behavior as Ref<T>::delete_object.

An application needs to initialize and manage the transient members of a persistent object as the object enters and exits the application cache. Memory may need to be allocated and deallocated when these events occur, for example. The d_activate function is called when an object enters the application cache, and d_deactivate is called

when the object exits the application cache. Normally C++ code uses the constructor and destructor to perform initialization and destruction, but in an ODMG implementation the constructor only gets called when an object is first created and the destructor is called at the point the object is deleted from the database. The following diagram depicts the calls made throughout the lifetime of an object.



The object first gets initialized by the constructor. At the point the object exits the application cache, the d_deactivate function gets called. When the object reenters the application cache, d_activate gets called. This may get repeated many times, as the object moves in and out of an application cache. Eventually, the object gets deleted, in which case only the destructor gets called, not d_deactivate.

5.3.5 Reference Classes

Objects may refer to other objects through a smart pointer or reference called a d_Ref. A d_Ref<T> is a reference to an instance of type T. There is also a d_Ref_Any class defined that provides a generic reference to any type.

A d_Ref is a template class defined as follows:

Definition:

```
template <class T> class d_Ref {
public:
                        d_Ref();
                        d_Ref(T *fromPtr);
                        d_Ref(const d_Ref<T> &);
                        d_Ref(const d_Ref_Any &);
                        ~d_Ref();
                        operator d_Ref_Any() const;
    d_Ref<T> &          operator=(T *);
    d_Ref<T> &          operator=(const d_Ref<T>&);
```

```
    void            clear();
    T *             operator->() const;     // dereference the reference
    T &             operator*() const;
    T *             ptr() const;
    void            delete_object();    // delete referred object from memory
                                        // and the database
// boolean predicates to check reference
    d_Boolean       operator!() const;
    d_Boolean       is_null() const;
// do these d_Refs and pointers refer to the same objects?
friend  d_Boolean   operator==(const d_Ref<T> &refL, const d_Ref<T> &refR );
friend  d_Boolean   operator==(const d_Ref<T> &refL, const T *ptrR );
friend  d_Boolean   operator==(const T *ptrL, const d_Ref<T> &refR );
friend  d_Boolean   operator==(const d_Ref<T> &L, const d_Ref_Any &R);
friend  d_Boolean   operator!=(const d_Ref<T> &refL, const d_Ref<T> &refR);
friend  d_Boolean   operator!=(const d_Ref<T> &refL, const T *ptrR);
friend  d_Boolean   operator!=(const T *ptrL, const d_Ref<T> &refR);
friend  d_Boolean   operator!=(const d_Ref<T> &refL, const d_Ref_Any &anyR);
};
```

References in many respects behave like C++ pointers but provide an additional mechanism that guarantees integrity in references to persistent objects. Although the syntax for declaring a d_Ref is different than for declaring a pointer, the usage is, in most cases, the same due to overloading; e.g., d_Refs may be dereferenced with the * operator, assigned with the = operator, etc. A d_Ref to a class may be assigned to a d_Ref to a superclass. d_Refs may be subclassed to provide specific referencing behavior.

There is one anomaly that results from the ability to do conversions between d_Ref<T> and d_Ref_Any. The following code will compile without error, and a d_Error exception object of kind d_Error_TypeInvalid is thrown at run-time versus statically at compile time. Suppose that X and Y are two unrelated classes:

```
d_Ref<X>    x;
d_Ref<Y>    y(x);
```

The initialization of y via x will be done via a conversion to d_Ref_Any. One should avoid such initializations in their application.

The pointer or reference returned by operator–> or operator * is only valid until either the d_Ref is deleted, the end of the outermost transaction, or until the object it points to is deleted. The pointer returned by ptr is only valid until the end of the outermost transaction or until the object it points to is deleted. The value of a d_Ref after a transaction commit or abort is undefined. If an attempt is made to dereference a null

d_Ref<T>, a d_Error exception object of kind d_Error_RefNull is thrown. Calling delete_object with a null d_Ref is silently ignored, as it is with a pointer in C++.

The following template class allows one to specify a to-one relationship to a class T:

```
template <class T, const char *Member> class d_Rel_Ref : public d_Ref<T> {  };
```

The template d_Rel_Ref<T,M> supports the same interface as d_Ref<T>. Implementations will redefine some functions to provide support for referential integrity.

A class d_Ref_Any is defined to support a reference to any type. Its primary purpose is to handle generic references and allow conversions of d_Refs in the type hierarchy. A d_Ref_Any object can be used as an intermediary between any two types d_Ref<X> and d_Ref<Y> where X and Y are different types. A d_Ref<T> can always be converted to a d_Ref_Any; there is a function to perform the conversion in the d_Ref<T> template. Each d_Ref<T> class has a constructor and assignment operator that takes a reference to a d_Ref_Any.

The d_Ref_Any class is defined as follows:

Definition:

```
class d_Ref_Any {
public:
                        d_Ref_Any();
                        d_Ref_Any(const d_Ref_Any &);
                        d_Ref_Any(d_Object *);
                        ~d_Ref_Any();
        d_Ref_Any &     operator=(const d_Ref_Any &);
        d_Ref_Any &     operator=(d_Object *);
        void            clear();
        void            delete_object();

// boolean predicates checking to see if value is null or not
        d_Boolean       operator!() const;
        d_Boolean       is_null() const;

friend  d_Boolean       operator==(const d_Ref_Any &, const d_Ref_Any &);
friend  d_Boolean       operator==(const d_Ref_Any &, const d_Object *);
friend  d_Boolean       operator==(const d_Object *, const d_Ref_Any &);
friend  d_Boolean       operator!=(const d_Ref_Any &, const d_Ref_Any &);
friend  d_Boolean       operator!=(const d_Ref_Any &, const d_Object *);
friend  d_Boolean       operator!=(const d_Object *, const d_Ref_Any &);
};
```

The operations defined on d_Ref<T> that are not dependent on a specific type T have been provided in the d_Ref_Any class.

5.3.6 Collection Classes

Collection templates are provided to support the representation of a collection whose elements are of an arbitrary type. A conforming implementation must support at least the following subtypes of d_Collection:

- d_Set
- d_Bag
- d_List
- d_Varray
- d_Dictionary

The C++ class definitions for each of these types are defined in the subsections that follow. Iterators are defined as a final subsection.

The following discussion uses the d_Set class in its explanation of collections, but the description applies for all concrete classes derived from d_Collection.

Given an object of type T, the declaration

```
d_Set<T> s;
```

defines a d_Set collection whose elements are of type T. If this set is assigned to another set of the same type, both the d_Set object itself and each of the elements of the set are copied. The elements are copied using the copy semantics defined for the type T. A common convention will be to have a collection that contains d_Refs to persistent objects—for example,

```
d_Set<d_Ref<Professor> > faculty;
```

The d_Ref class has shallow copy semantics. For a d_Set<T>, if T is of type d_Ref<C> for some persistence-capable class C, only the d_Ref objects are copied, not the C objects that the d_Ref objects reference.

This holds in any scope; in particular, if s is declared as a member inside a class, the set itself will be embedded inside an instance of this class. When an object of this enclosing class is copied into another object of the same enclosing class, the embedded set is copied, too, following the copy semantics defined above. This must be differentiated from the declaration

```
d_Ref<d_Set<T> >  ref_set;
```

which defines a reference to a d_Set. When such a reference is defined as a property of a class, that means that the set itself is an independent object that lies outside an instance of the enclosing class. Several objects may then share the same set, since copying an object will not copy the set, but just the reference to it. These are illustrated in Figure 5-2.

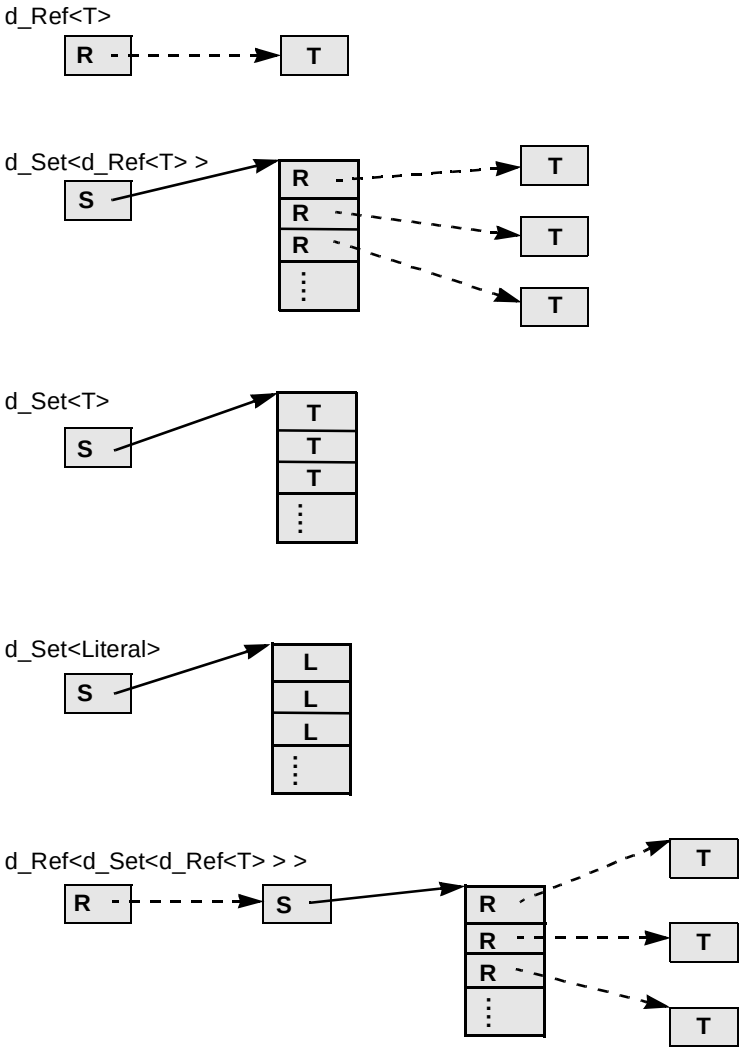


Figure 5-2. Collections, Embedded and with d_Ref

Collection elements may be of any type. Every type T that will become an element of a given collection must support the following operations:

```
class T {
public:
        T();
        T(const T &);
        ~T();
```

```
    T &       operator=(const T &);
friend int    operator==(const T&, const T&);
};
```

This is the complete set of functions required for defining the copy semantics for a given type. For types requiring ordering, the following operation must also be provided:

```
friend d_Boolean    operator<(const T&, const T&);
```

Note that the C++ compiler will automatically generate a copy constructor and assignment operator if the class designer does not declare one. Note that the d_Ref<T> class supports these operations, except for operator<.

Collections of literals, including both atomic and structured literals, are defined as part of the standard. This includes both primitive and user-defined types; e.g., d_Set<int>, d_Set<struct time_t> will be defined with the same behavior.

Figure 5-2 illustrates various types involving d_Sets, d_Refs, a literal type L (int for example) and a persistent class T. The d_Set object itself is represented by a box that then refers to a set of elements of the specified type. A solid arrow is used to denote containment of the set elements within the d_Set object. d_Refs have a dashed arrow pointing to the referenced object of type T.

5.3.6.1 Class d_Collection

Class d_Collection is an abstract class in C++ and cannot have instances. It is derived from d_Object, allowing instances of concrete classes derived from d_Collection to be stand-alone persistent objects.

Definition:

```
template <class T> class d_Collection : public d_Object {
public:
virtual                 ~d_Collection();
    d_Collection<T> &   assign_from(const d_Collection<T> &);
friend d_Boolean        operator==( const d_Collection<T> &cL,
                                    const d_Collection<T> &cR);
friend d_Boolean        operator!=( const d_Collection<T> &cL,
                                    const d_Collection<T> &cR);
    unsigned long       cardinality() const;
    d_Boolean           is_empty() const;
    d_Boolean           is_ordered() const;
    d_Boolean           allows_duplicates() const;
    d_Boolean           contains_element(const T &element) const;
```

```
        void                insert_element(const T &elem);
        void                remove_element(const T &elem);
        void                remove_all();
        void                remove_all(const T &elem);
        d_Iterator<T>       create_iterator() const;
        d_Iterator<T>       begin() const;
        d_Iterator<T>       end() const;
        T                   select_element(const char *OQL_predicate) const;
        d_Iterator<T>       select(const char * OQL_predicate) const;
        int                 query(d_Collection<T>&, const char *OQL_pred) const;
        d_Boolean           exists_element(const char* OQL_predicate) const;
    protected:
                            d_Collection(const d_Collection<T> &);
        d_Collection<T> &   operator=(const d_Collection<T> &);
                            d_Collection();
    };
```

Note that the d_Collection class provides the operation assign_from in place of operator= because d_Collection assignment is relatively expensive. This will prevent the often gratuitous use of assignment with collections.

The member function remove_element() removes one element that is equal to the argument from the collection. For ordered collections this will be the element that is equal to the value that would be first encountered if performing a forward iteration of the collection. The remove_all() function removes all of the elements from the collection. The remove_all(const T&) function removes all of the elements in the collection that are equal to the supplied value.

The destructor is called for an element whenever the element is removed from the collection. This also applies when the collection itself is assigned to or removed. If the element type of the collection is d_Ref<T> for some class T, the destructor of d_Ref<T> is called, but not the destructor of T.

The equality can be evaluated of two instances of any collection class derived from d_Collection that have the same element type. When comparing an instance of d_Set to either an instance of d_Set, d_Bag, d_List, or d_Varray, they are only equal if they have the same cardinality and the same elements. When comparing an instance of d_Bag to an instance of d_Bag, d_List, or d_Varray, they are equal if they have the same cardinality and the same number of occurrences of each element value. The ordering of the elements in the d_List or d_Varray does not matter; they are treated like a d_Bag. An instance of d_List or d_Varray is equal to another instance of d_List or d_Varray if they have the same cardinality and the element at each position is equal.

The create_iterator function returns an iterator pointing at the first element in the collection. The begin and end functions are supplied for compatibility with the C++ Standard Template Library (STL) algorithms. The function begin returns an iterator positioned at the first element of iteration. As defined in STL, the end function returns an iterator value that is "past the end" of iteration and is not dereferenceable.

5.3.6.2 Class d_Set

A d_Set<T> is an unordered collection of elements of type T with no duplicates.

Definition:

```
template <class T> class d_Set : public d_Collection<T> {
public:
                        d_Set();
                        d_Set(const d_Set<T> &);
                        ~d_Set();
      d_Set<T> &        operator=(const d_Set<T> &);
      d_Set<T> &        union_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        union_with(const d_Set<T> &s2);
      d_Set<T> &        operator+=(const d_Set<T> &s2);          // union_with
      d_Set<T>          create_union(const d_Set<T> &s) const;
friend  d_Set<T>        operator+(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Set<T> &        intersection_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        intersection_with(const d_Set<T> &s2);
      d_Set<T> &        operator*=(const d_Set<T> &s2);          // intersection_with
      d_Set<T>          create_intersection(const d_Set<T> &s) const;
friend  d_Set<T>        operator*(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Set<T> &        difference_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        difference_with(const d_Set<T> &s2);
      d_Set<T> &        operator-=(const d_Set<T> &s2);          // difference_with
      d_Set<T>          create_difference(const d_Set<T> &s) const;
friend  d_Set<T>        operator-(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Boolean         is_subset_of(const d_Set<T> &s2) const;
      d_Boolean         is_proper_subset_of(const  d_Set<T> &s2) const;
      d_Boolean         is_superset_of(const d_Set<T> &s2) const;
      d_Boolean         is_proper_superset_of(const d_Set<T> &s2) const;
};
```

Note that all operations defined on type d_Collection are inherited by type d_Set, e.g., insert_element, remove_element, select_element, and select.

Examples:

- creation:

```
d_Database db;                  // assume we open a database
d_Ref<Professor> Guttag; // assume we set this to a professor
d_Ref<d_Set<d_Ref<Professor> > > my_profs =
                                        new(&db) d_Set<d_Ref<Professor> >;
```

- insertion:

```
my_profs->insert_element(Guttag);
```

- removal:

```
my_profs->remove_element(Guttag);
```

- deletion:

```
my_profs.delete_object();
```

For each of the set operations (union, intersection, and difference) there are three ways of computing the resulting set. These will be explained using the union operation. Each one of the union functions has two set operands and computes their union. They vary in how the set operands are passed and how the result is returned, to support different interface styles. The union_of function is a member function that has two arguments, which are references to d_Set<T>. It computes the union of the two sets and places the result in the d_Set object with which the function was called, removing the original contents of the set. The union_with function is also a member and places its result in the object with which the operation is invoked, removing its original contents. The difference is that union_with uses its current set contents as one of the two operands being unioned, thus requiring only one operand passed to the member function. Both union_of and union_with return a reference to the object with which the operation was invoked. The union_with function has a corresponding operator+= function defined. On the other hand, create_union creates and returns a new d_Set instance by value that contains the union, leaving the two original sets unaltered. This function also has a corresponding operator+ function defined.

The following template class allows one to specify an unordered to-many relationship with a class T:

```
template <class T, const char *M> class d_Rel_Set : public d_Set<d_Ref<T> > { }
```

The template d_Rel_Set<T,M> supports the same interface as d_Set<d_Ref<T> >. Implementations will redefine some functions in order to support referential integrity.

5.3.6.3 Class d_Bag

A d_Bag<T> is an unordered collection of elements of type T that does allow for duplicate values.

Definition:

```
template <class T> class d_Bag : public d_Collection<T> {
public:
                        d_Bag();
                        d_Bag(const d_Bag<T> &);
                        ~d_Bag();
    d_Bag<T> &          operator=(const d_Bag<T> &);
    unsigned long       occurrences_of(const T &element) const;
    d_Bag<T> &          union_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          union_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator+=(const d_Bag<T> &b2); // union_with
    d_Bag<T>            create_union(const d_Bag<T> &b) const;
friend  d_Bag<T>        operator+(const d_Bag<T> &b1, const d_Bag<T> &b2);
    d_Bag<T> &          intersection_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          intersection_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator*=(const d_Bag<T> &b2);   // intersection_with
    d_Bag<T>            create_intersection(const d_Bag<T>&b) const;
friend  d_Bag<T>        operator*(const d_Bag<T> &b1, const d_Bag<T> &b2);
    d_Bag<T> &          difference_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          difference_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator-=(const d_Bag<T> &b2); // difference_with
    d_Bag<T>            create_difference(const d_Bag<T> &b) const;
friend  d_Bag<T>        operator-(const d_Bag<T> &b1, const d_Bag<T> &b2);
};
```

The union, intersection, and difference operations are described in the section above on the d_Set class.

5.3.6.4 Class d_List

A d_List<T> is an ordered collection of elements of type T and does allow for duplicate values. The beginning d_List index value is 0, following the convention of C and C++.

Definition:

```
template <class T> class d_List : public d_Collection<T> {
public:
                        d_List();
                        d_List(const d_List<T> &);
                        ~d_List();
```

```
        d_List<T> &     operator=(const d_List<T> &);
        T               retrieve_first_element() const;
        T               retrieve_last_element() const;
        void            remove_first_element();
        void            remove_last_element();
        T               operator[](unsigned long position) const;
        d_Boolean       find_element(const T &element,
                                     unsigned long &position) const;
        T               retrieve_element_at(unsigned long position) const;
        void            remove_element_at(unsigned long position);
        void            replace_element_at(const T &element,
                                           unsigned long position);
        void            insert_element_first(const T &element);
        void            insert_element_last(const T &element);
        void            insert_element_after(const T & element,
                                           unsigned long position);
        void            insert_element_before(const T &element,
                                             unsigned long position);
        d_List<T>       concat(const d_List<T> &listR) const;
    friend  d_List<T>   operator+(const d_List<T> &listL, const d_List<T> &listR);
        d_List<T> &     append(const d_List<T> &listR);
        d_List<T> &     operator+=(const d_List<T> &listR);
    };
```

The insert_element function (inherited from d_Collection <T>) inserts a new element at the end of the list. The subscript operator (operator[]) has the same semantics as the member function retrieve_element_at. The concat function creates a new d_List<T> that contains copies of the elements from the original list, followed by copies of the elements from listR. The original lists are not affected. Similarly, operator+ creates a new d_List<T> that contains copies of the elements in listL and listR and does not change either list. The append function and operator+= both copy elements from listR and adds them after the last element of the list. The modified list is returned as the result.

The d_Rel_List<T, Member> template is used for representing relationships that are positional in nature:

```
template <class T, const char *M> class d_Rel_List : public d_List<d_Ref<T> > { };
```

The template d_Rel_List<T,M> has the same interface as d_List<d_Ref<T> >.

5.3.6.5 Class Array

The Array type defined in Section 2.3.6 is implemented by the built-in array defined by the C++ language. This is a single-dimension, fixed-length array.

5.3.6.6 Class d_Varray

A d_Varray<T> is a one-dimensional array of varying length containing elements of type T. The beginning d_Varray index value is 0, following the convention of C and C++.

Definition:

```
template <class T> class d_Varray : public d_Collection<T> {
public:
                        d_Varray();
                        d_Varray(unsigned long length);
                        d_Varray(const d_Varray<T> &);
                        ~d_Varray();
    d_Varray<T> &       operator=(const d_Varray<T> &);
    void                resize(unsigned long length);
    T                   operator[](unsigned long index) const;
    d_Boolean           find_element(const T &element,
                                     unsigned long &index) const;
    T                   retrieve_element_at(unsigned long index) const;
    void                remove_element_at(unsigned long index);
    void                replace_element_at(const T &element,
                                           unsigned long index);
};
```

The insert_element function (inherited from d_Collection <T>) inserts a new element by increasing the d_Varray length by one and placing the new element at this new position in the d_Varray.

Examples:

```
d_Varray<d_Double>  vector(1000);
vector.replace_element_at(3.14159, 97);
vector.resize(2000);
```

5.3.6.7 Class d_Dictionary

The d_Dictionary<K,V> class is an unordered collection of key-value pairs, with no duplicate keys. A key-value pair is represented by an instance of d_Association<K,V>.

```
template <class K, class V> class d_Association
{
public:
    K       key;
    V       value;
            d_Association(const K &k, const V &v) : key(k), value(v) { }
};
```

The d_Dictionary<K,V> inherits from class d_Collection<T> and thus supports all of its base class operations. The insert_element, remove_element, and contains_element operations inherited from d_Collection<T> are valid for d_Dictionary<K,V> types when a d_Association<K,V> is specified as the argument. The contains_element function returns true if both the key and value specified in the d_Association parameter are contained in the dictionary.

```
template <class K, class V>
class d_Dictionary : public d_Collection<d_Association<K,V> > {
public:
                            d_Dictionary();
                            d_Dictionary(const d_Dictionary<K,V> &);
                            ~d_Dictionary();
    d_Dictionary<K,V> &     operator=(const d_Dictionary<K,V> &);
    void                    bind(const K&, const V&);
    void                    unbind(const K&);
    V                       lookup(const K&) const;
    d_Boolean               contains_key(const K&) const;
};
```

Iterating over a d_Dictionary<K,V> object will result in the iteration over a sequence of d_Association<K,V> instances. Each get_element operation, executed on an instance of d_Iterator<T>, returns an instance of d_Association<K,V>. If insert_element inherited from d_Collection is called and a duplicate key is found, its value is replaced with the new value passed to insert_element. The bind operation works the same as insert_element except the key and value are passed separately. When remove_element inherited from d_Collection is called, both the key and value must be equal for the element to be removed. An exception error of d_Error_ElementNotFound is thrown if the d_Association is not found in the dictionary. The function unbind removes the element with the specified key.

5.3.6.8 Class d_Iterator

A template class, d_Iterator<T>, defines the generic behavior for iteration. All iterators use a consistent protocol for sequentially returning each element from the collection

over which the iteration is defined. A template class has been used to give us type-safe iterators, i.e., iterators that are guaranteed to return an instance of the type of the element of the collection over which the iterator is defined. Normally, an iterator is initialized by the create_iterator method on a collection class.

The template class d_Iterator<T> is defined as follows:

```
template <class T> class d_Iterator {
public:
                        d_Iterator();
                        d_Iterator(const d_Iterator<T> &);
                        ~d_Iterator();
      d_Iterator<T> &   operator=(const d_Iterator<T> &);
friend  d_Boolean       operator==(const d_Iterator<T> &, const d_Iterator<T> &);
friend  d_Boolean       operator!=(const d_Iterator<T> &, const d_Iterator<T> &);
      void              reset();
      d_Boolean         not_done() const;
      void              advance();
      d_Iterator<T> &   operator++();
      d_Iterator<T>     operator++(int);
      d_Iterator<T> &   operator--();
      d_Iterator<T>     operator--(int);
      T                 get_element() const;
      T                 operator*() const;
      void              replace_element(const T &);
      d_Boolean         next(T &objRef);
};
```

When an iterator is constructed, it is either initialized with another iterator or set to null. When an iterator is constructed via the create_iterator function defined in d_Collection, the iterator is initialized to point to the first element, if there is one. Iterator assignment is also supported. A reset function is provided to reinitialize the iterator to the start of iteration for the same collection. The replace_element function can only be used with d_List or d_Varray.

The not_done function allows one to determine whether there are any more elements in the collection to be visited in the iteration. It returns 1 if there are more elements and 0 if iteration is complete. The advance function moves the iterator forward to the next element in the collection. The prefix and postfix forms of the increment operator ++ have been overloaded to provide an equivalent advance operation. One can also move backward through the collection by using the decrement operator --. However, using the -- decrement operator on an iterator of an unordered collection will throw a d_Error exception object of kind d_Error_IteratorNotBackward. If an attempt is made

to either advance an iterator once it has already reached the end of a collection or move backward once the first element has been reached, a d_Error exception object of kind d_Error_IteratorExhausted is thrown. An attempt to use an iterator with a different collection than the collection it is associated with causes a d_Error exception object of kind d_Error_IteratorDifferentCollections to be thrown.

The get_element function and operator* return the value of the current element. If there is no current element, a d_Error exception object of kind d_Error_IteratorExhausted is thrown. There would be no current element if iteration had been completed (not_done return of 0) or if the collection had no elements.

The next function provides a facility for checking the end of iteration, advancing the iterator and returning the current element, if there is one. Its behavior is as follows:

```
template <class T> d_Boolean d_Iterator<T>::next(T &objRef)
{
    if( !not_done() ) return 0;   // no more elements, return false
    objRef = get_element();       // assign current element into output parameter
    advance();                    // advance to the next element
    return 1;                     // return true, that there is a next element
}
```

These operations allow for two styles of iteration, using either a while or for loop.

Example:

Given the class Student, with extent students:

```
(1)  d_Iterator<d_Ref<Student> >  iter = students.create_iterator();
     d_Ref<Student>  s;
(2)  while( iter.next(s) ) {
         ....
     }
```

Note that calling get_element after calling next will return a different element (the next element, if there is one). This is due to the fact that next will access the current element and then advance the iterator before returning.

Or equivalently with a for loop:

```
(3)  d_Iterator<d_Ref<Student> > iter = students.create_iterator();
(4)  for( ; iter.not_done(); ++iter) {
(5)      d_Ref<Student> s = iter.get_element();
         ....
     }
```

Statement (1) defines an iterator iter that ranges over the collection students. Statement (2) iterates through this collection, returning a d_Ref to a Student on each successive call to next, binding it to the loop variable s. The body of the while statement is then executed once for each student in the collection students. In the for loop (3) the iterator is initialized, iteration is checked for completion, and the iterator is advanced. Inside the for loop the get_element function can be called to get the current element.

5.3.6.9 Collections and the Standard Template Library

The C++ Standard Template Library (STL) provides an extensible set of containers, i.e., collections and algorithms that work together in a seamless way. The ODMG C++ language binding extends STL with persistence-capable versions of STL's container classes, each of which may be operated on by all template algorithms in the same manner as transient containers. A conforming implementation must provide at least the following persistence-capable STL container types, derived from d_Object:

- d_set
- d_multiset
- d_vector
- d_list
- d_map
- d_multimap

The names of these containers have the ODMG prefix (d_) and have interfaces that correspond to the STL set, multiset, vector, list, map, and multimap containers, respectively.

The C++ STL may be used to operate on the collection classes derived from d_Collection defined in the C++ OML. STL algorithms traverse container data structures using iterator objects, which are compatible with the d_Iterator<T> objects defined in the C++ OML. Specifically, d_Iterator<T> objects conform to the STL specification of constant iterators of the bidirectional_iterator category, though an implementation may provide more powerful iterators in some circumstances.

5.3.7 Transactions

Transaction semantics are defined in the object model explained in Chapter 2.

Transactions can be started, committed, aborted, and checkpointed. It is important to note that *all access, creation, modification, and deletion of persistent objects must be done within a transaction.*

Transactions are implemented in C++ as objects of class d_Transaction. The class d_Transaction defines the operation for starting, committing, aborting, and checkpointing transactions. These operations are

```
class d_Transaction {
public:
                        d_Transaction();
                        ~d_Transaction();
    void                begin();
    void                commit();
    void                abort();
    void                checkpoint();
// Thread operations
    void                join();
    void                leave();
    d_Boolean           is_active() const;
static d_Transaction *  current();
private:
                        d_Transaction(const d_Transaction &);
    d_Transaction &     operator=(const d_Transaction &);
};
```

Transactions must be explicitly created and started; they are not automatically started on database open, upon creation of a d_Transaction object, or following a transaction commit or abort.

The begin function starts a transaction. Calling begin multiple times on the same transaction object, without an intervening commit or abort, causes a d_Error exception object of kind d_Error_TransactionOpen to be thrown on second and subsequent calls. If a call is made to commit, checkpoint, or abort on a transaction object and a call had not been initially made to begin, a d_Error exception object of kind d_Error_TransactionNotOpen is thrown.

Calling commit commits to the database all persistent objects modified (including those created or deleted) within the transaction and releases any locks held by the transaction. Implementations may choose to maintain the validity of d_Refs to persistent objects across transaction boundaries. The commit operation does not delete the transaction object.

Calling checkpoint commits objects modified within the transaction since the last checkpoint to the database. The transaction retains all locks it held on those objects at the time the checkpoint was invoked. All d_Refs and pointers remain unchanged.

Calling abort aborts changes to objects and releases the locks, and does not delete the transaction object.

The destructor aborts the transaction if it is active.

The boolean function is_active returns d_True if the transaction is active; otherwise it returns d_False.

In the current standard, transient objects are not subject to transaction semantics. Committing a transaction does not remove transient objects from memory. Aborting a transaction does not restore the state of modified transient objects.

d_Transaction objects are not long-lived (beyond process boundaries) and cannot be stored to the database. This means that transaction objects may not be made persistent and that the notion of "long transactions" is not defined in this specification.

In summary the rules that apply to object modification (necessarily, during a transaction) are

1. Changes made to persistent objects within a transaction can be "undone" by aborting the transaction.
2. Transient objects are standard C++ objects.
3. Persistent objects created within the scope of a transaction are handled consistently at transaction boundaries: stored to the database and removed from memory (at transaction commit) or deleted (as a result of a transaction abort).

A thread must explicitly create a transaction object or associate itself with an existing transaction object by calling join. The member function join attaches the caller's thread to the transaction and the thread is detached from any other transaction it may be associated with. Calling begin on a transaction object without doing a prior join implicitly joins the transaction to the calling thread. All subsequent operations by the thread, including reads, writes, and implicit lock acquisitions, are done under the thread's current transaction.

Calling leave detaches the caller's thread from the d_Transaction instance without attaching the thread to another transaction. The static function current can be called to access the current d_Transaction object that the thread is associated with; null is returned if the thread is not associated with a transaction.

If a transaction is associated with multiple threads, all of these threads are affected by any data operations or transaction operations (begin, commit, checkpoint, abort). Concurrency control on data among threads is up to the client program in this case. In contrast, if threads use separate transactions, the database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers must not pass objects from one thread to another running under a different transaction; ODMG does not define the results of doing this.

There are three ways in which threads can be used with transactions:

1. An application program may have exactly one thread doing database operations, under exactly one transaction. This is the simplest case, and it represents the vast majority of database applications today. Other applications on

separate machines or in separate address spaces may access the same database under separate transactions. A thread can create multiple instances of d_Transaction and can alternate between them by calling join.

2. There may be multiple threads, each with its own separate transaction. This is useful for writing a service accessed by multiple clients on a network. The database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers must not pass objects from one thread to another thread running under a different transaction; ODMG does not define the results of doing this.

3. Multiple threads may share one or more transactions. Using multiple threads per transaction is only recommended for sophisticated programming because concurrency control must be performed by the application.

5.3.8 d_Database Operations

There is a predefined type d_Database. It supports the following methods:

```
class d_Database {
public:
static d_Database * const transient_memory;
    enum access_status { not_open, read_write, read_only, exclusive };
                        d_Database();
    void                open(    const char * database_name,
                                 access_status status = read_write);
    void                close();
    void                set_object_name(const d_Ref_Any &theObject,
                                 const char* theName);
    void                rename_object(   const char * oldName,
                                 const char * newName);
    d_Ref_Any           lookup_object(const char * name) const;
private:
                        d_Database(const d_Database &);
    d_Database &        operator=(const d_Database &);
};
```

The database object, like the transaction object, is transient. Databases cannot be created programmatically using the C++ OML defined by this standard. Databases must be opened before starting any transactions that use the database, and closed after ending these transactions.

To open a database, use d_Database::open, which takes the name of the database as its argument. This initializes the instance of the d_Database object.

```
database–>open("myDB");
```

Method open locates the named database and makes the appropriate connection to the database. You must open a database before you can access objects in that database. Attempts to open a database when it has already been opened will result in the throwing of a d_Error exception object of kind d_Error_DatabaseOpen. Extensions to the open method will enable some ODBMSs to implement default database names and/or implicitly open a default database when a database session is started. Some ODBMSs may support opening logical as well as physical databases. Some ODBMSs may support being connected to multiple databases at the same time.

To close a database, use d_Database::close:

```
database->close();
```

Method close does appropriate clean-up on the named database connection. After you have closed a database, further attempts to access objects in the database will cause a d_Error exception object of kind d_Error_DatabaseClosed to be thrown. The behavior at program termination if databases are not closed or transactions are not committed or aborted is undefined.

The *name* methods allow manipulating names of objects. The set_object_name method assigns a character string name to the object referenced. If the string supplied as the name argument is not unique within the database, a d_Error exception object of kind d_Error_NameNotUnique will be thrown. Each call to set_object_name for an object adds an additional name to the object. If a value of 0 is passed as the second parameter to set_object_name, all of the names associated with the object are removed.

The rename_object method changes the name of an object. If the new name is already in use, a d_Error exception object of kind d_Error_NameNotUnique will be thrown and the old name is retained. A name can be removed by passing 0 as the second parameter to rename_object. Names are not automatically removed when an object is deleted. If a call is made to lookup_object with the name of a deleted object, a null d_Ref_Any is returned. Previous releases removed the names when the object was deleted.

An object is accessed by name using the d_Database::lookup_object member function. A null d_Ref_Any is returned if no object with the name is found in the database.

Example:

```
d_Ref<Professor> prof = myDatabase->lookup_object("Newton");
```

If a Professor instance named “Newton” exists, it is retrieved and a d_Ref_Any is returned by lookup_object. The d_Ref_Any return value is then used to initialize prof. If the object named “Newton” is not an instance of Professor or a subclass of Professor, a d_Error exception object of kind d_Error_TypeInvalid is thrown during this initialization.

If the definition of a class in the application does not match the database definition of the class, a d_Error exception object of kind d_Error_DatabaseClassMismatch is thrown.

5.3.9 Class d_Extent<T>

The class d_Extent<T> provides an interface to the extent of a persistence-capable class T in the C++ binding.

d_Extent provides nearly the same interface as the d_Collection class.

```
template <class T> class d_Extent
{
public:
                        d_Extent(const d_Database* base,
                                d_Boolean include_subclasses = d_True);
    virtual             ~d_Extent();
    unsigned long       cardinality() const;
    d_Boolean           is_empty() const;
    d_Boolean           allows_duplicates() const;
    d_Boolean           is_ordered() const;
    d_Iterator<T>       create_iterator() const;
    d_Iterator<T>       begin() const;
    d_Iterator<T>       end() const;
    d_Ref<T>            select_element(const char * OQL_pred) const;
    d_Iterator<T>       select(const char * OQL_pred) const;
    int                 query(d_Collection<d_Ref<T> > &,
                              const char* OQL_pred) const;
    d_Boolean           exists_element(const char * OQL_pred) const;
protected:
                        d_Extent(const d_Extent<T> &);
    d_Extent<T> &       operator=(const d_Extent<T> &);
};
```

The database schema definition contains a parameter for each persistent class specifying whether the ODBMS should maintain an extent for the class. This parameter can be set using the schema API or a database tool that enables specification of the schema.

The content of a d_Extent<T> is automatically maintained by the ODBMS. The d_Extent class therefore has neither insert nor remove methods. d_Extents themselves are not persistence-capable and cannot be stored in the database. This explains why d_Extent is not derived from d_Collection; since d_Collection is in turn derived from

d_Object, this would imply that extents are also persistence-capable. However, semantically d_Extent is equivalent to d_Set.

If a user wants to maintain an extent they can define a d_Set<d_Ref<T> > that is stored in the database, as in the example in Section 5.6.

The class d_Extent supports polymorphism when the constructor argument include_subclasses is a true value. If type B is a subtype of A a d_Extent for A includes all instances of A and B.

The association of a d_Extent to a type is performed by instantiating the template with the appropriate type. Every d_Extent instance must be associated with a database by passing a d_Database pointer to the constructor.

```
d_Extent<Person>    PersonExtent(database);
```

Passing the database pointer to the constructor instead of operator new (as with d_Object) allows the user to instantiate a d_Extent instance on the stack. If no extent has been defined in the database schema for the class, an exception is thrown.

Comparison operators like operator== and operator!= or the subset and superset methods of d_Set do not make sense for d_Extent, since all instances of a d_Extent for a given type have the same content.

5.3.10 Exceptions

Instances of d_Error contain state describing the cause of the error. This state is composed of a number, representing the kind of error, and optional additional information. This additional information can be appended to the error object by using operator<<. If the d_Error object is caught, more information can be appended to it if it is to be thrown again. The complete state of the object is returned as a human-readable character string by the what function.

The d_Error class is defined as follows:

```
class d_Error : public exception {
public:
    typedef d_Long    kind;
                      d_Error();
                      d_Error(const d_Error &);
                      d_Error(kind the_kind);
                      ~d_Error();
    kind              get_kind();
    void              set_kind(kind the_kind);
    const char *      what() const throw();

    d_Error &         operator<<(d_Char);
    d_Error &         operator<<(d_Short);
```

```
        d_Error &       operator<<(d_UShort);
        d_Error &       operator<<(d_Long);
        d_Error &       operator<<(d_ULong);
        d_Error &       operator<<(d_Float);
        d_Error &       operator<<(d_Double);
        d_Error &       operator<<(const char *);
        d_Error &       operator<<(const d_String &);
        d_Error &       operator<<(const d_Error &);
};
```

The null constructor initializes the kind property to d_Error_None. The class d_Error is responsible for releasing the string that is returned by the member function what.

The following constants are defined for error kinds used in this standard. Note that each of these names has a prefix of d_Error_.

Error Name	Description
None	No error has occurred.
DatabaseClassMismatch	The definition of a class in the application does not match the database definition of the class.
DatabaseClassUndefined	The database does not have the schema information about a class.
DatabaseClosed	The database is closed; objects cannot be accessed.
DatabaseOpen	The database is already open.
DateInvalid	An attempt was made to set a d_Date object to an invalid value.
ElementNotFound	An attempt was made to access an element that is not in the collection.
IteratorDifferentCollections	An iterator, passed to a member function of a collection, is not associated with the collection.
IteratorExhausted	An attempt was made to either advance an iterator once it already reached the end of a collection or move backward once the first element has been reached.
IteratorNotBackward	An attempt has been made to iterate backward with either a d_Set<T> or d_Bag<T>.
NameNotUnique	An attempt was made to associate an object with a name that is not unique in the database.

Error Name	Description
PositionOutOfRange	A position within a collection has been supplied that exceeds the range of the index (0, cardinality - 1).
QueryParameterCountInvalid	The number of arguments used to build a query with the d_OQL_Query object does not equal the number of arguments supplied in the query string.
QueryParameterTypeInvalid	Either the parameters specified in the query or the return value type does not match the types in the database.
RefInvalid	An attempt was made to dereference a d_Ref that references an object that does not exist.
RefNull	An attempt was made to dereference a null d_Ref.
TimeInvalid	An attempt has been made to set a d_Time object to an invalid value.
TimestampInvalid	An attempt has been made to set a d_Timestamp object to an invalid value.
MemberIsOfInvalidType	The second template argument of either d_Rel_Ref, d_Rel_Set, or d_Rel_List references a data member that is not of type d_Rel_Ref, d_Rel_Set, or d_Rel_List.
MemberNotFound	The second template argument of either d_Rel_Ref, d_Rel_Set, or d_Rel_List references a data member that does not exist in the referenced class.
TransactionNotOpen	A call has been made to either commit or abort without a prior call to begin.
TransactionOpen	d_Transaction::begin has been called multiple times on the same transaction object, without an intervening commit or abort.
TypeInvalid	A d_Ref<T> was initialized to reference an object that is not of type T or a subclass of T.

The following table indicates which functions throw these exceptions.

Error Name	Raised By
DatabaseClassMismatch	d_Database::lookup_object
DatabaseClassUndefined	d_Object::new
DatabaseClosed	d_Database::close
DatabaseOpen	d_Database::open
DateInvalid	Any method that alters the date value

Error Name	Raised By
IteratorExhausted	d_Iterator functions get_element, ++, --,operator *
NameNotUnique	d_Database::set_object_name
PositionOutOfRange	d_List::operator[ ], d_Varray::operator[ ]
QueryParameterCountInvalid	d_oql_execute
QueryParameterTypeInvalid	d_oql_execute
RefInvalid	d_Ref<T> functions operator–>, operator *
RefNull	d_Ref<T> functions operator–>, operator *
TimeInvalid	Any method that alters the time value
TimestampInvalid	Any method that alters the timestamp value
TransactionOpen	d_Transaction::begin
TypeInvalid	d_Ref constructor

5.4 C++ OQL

Chapter 4 outlined the Object Query Language. In this section the OQL semantics are mapped into the C++ language.

5.4.1 Query Method on Class Collection

The d_Collection class has a query member function whose signature is:

```
int query(d_Collection<T> &result, const char* predicate) const;
```

This function filters the collection using the predicate and assigns the result to the first parameter. It returns a code different from 0, if the query is not well formed. The predicate is given as a string with the syntax of the *where* clause of OQL. The predefined variable this is used inside the predicate to denote the current element of the collection to be filtered.

Example:

Given the class Student, as defined in Chapter 3, with extent referenced by Students, compute the set of students who take math courses:

```
d_Bag<d_Ref<Student> >  mathematicians;
Students–>query(mathematicians,
                    “exists s in this.takes: s.section_of.name = \“math\” ”);
```

5.4.2 d_oql_execute Function

An interface is provided to gain access to the complete functionality of OQL from a C++ program. There are several steps involved in the specification and execution of the OQL query. First, a query gets *constructed* via an object of type d_OQL_Query.

Once a query has been constructed, the query is *executed*. Once constructed, a query can be executed multiple times with different argument values.

The function to execute a query is called d_oql_execute; it is a free-standing template function, not part of any class definition:

```
template<class T> void      d_oql_execute(d_OQL_Query &query, T &result);
```

The first parameter, *query*, is a reference to a d_OQL_Query object specifying the query to execute. The second parameter, *result*, is used for returning the result of the query. The type of the query result must match the type of this second parameter, or a d_Error exception object of kind d_Error_QueryParameterTypeInvalid is thrown. Type checking of the input parameters according to their use in the query is done at runtime. Similarly, the type of the result of the query is checked. Any violation of type would cause a d_Error exception object of kind d_Error_QueryParameterTypeInvalid to be thrown. If the query returns a persistent object of type T, the function returns a d_Ref<T>. If the query returns a structured literal, the value of it is assigned to the value of the object or collection denoted by the *result* parameter.

If the result of the query is a large collection, a function d_oql_execute can be used. This function returns an iterator on the result collection instead of the collection itself. The behavior of this standalone function is exactly the same as the d_oql_execute function.

```
template<class T> void d_oql_execute (d_OQL_Query &q, d_Iterator<T> &results);
```

The << operator has been overloaded for d_OQL_Query to allow construction of the query. It concatenates the value of the right operand onto the end of the current value of the d_OQL_Query left operand. These functions return a reference to the left operand so that invocations can be cascaded.

Note that instances of d_OQL_Query contain either a partial or a complete OQL query. An ODMG implementation will contain ancillary data structures to represent a query both during its construction and once it is executed. The d_OQL_Query destructor will appropriately remove any ancillary data when the object gets deleted.

The d_OQL_Query class is defined as follows:

Definition:

```
class d_OQL_Query {
public:
                        d_OQL_Query();
                        d_OQL_Query(const char *s);
                        d_OQL_Query(const d_String &s);
                        d_OQL_Query(const d_OQL_Query &q);
                        ~d_OQL_Query();
```

```
    d_OQL_Query &       operator=(const d_OQL_Query &q);
    void                clear();

friend d_OQL_Query &    operator<<(d_OQL_Query &q, const char *s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_String &s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Char c);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Octet uc);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Short s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_UShort us);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, int i);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, unsigned int ui);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Long l);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_ULong ul);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Float f);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Double d);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Date &d);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Time &t);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Timestamp &);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Interval &i);
template<class T> friend d_OQL_Query &    operator<<(d_OQL_Query &q,
                                                     const d_Ref<T> &r);
template<class T> friend d_OQL_Query & operator<<(d_OQL_Query &q,
                                                     const d_Collection<T> &c);
};
```

Strings used in the construction of a query may contain parameters signified by the form *$i*, where *i* is a number referring to the i^{th} subsequent right operand in the construction of the query; the first subsequent right operand would be referred to as $1. If any of the $i are not followed by a right operand construction argument at the point d_oql_execute is called, a d_Error exception object of kind d_Error_QueryParameterCountInvalid is thrown. This exception will also be thrown if too many parameters are used in the construction of the query. If a query argument is of the wrong type, a d_Error exception object of kind d_Error_QueryParameterTypeInvalid is thrown.

The operation clear can be called to clear the values of any query parameters that have been provided to an instance of d_OQL_Query.

Once a query has been successfully executed via d_oql_execute, the arguments associated with the *$i* parameters are cleared and new arguments must be supplied. If any exceptions are thrown, the query arguments are not implicitly cleared and must be cleared explicitly by invoking clear. The original query string containing the *$i* parameters is retained across the call to d_oql_execute.

The d_OQL_Query copy constructor and assignment operator copy all the underlying data structures associated with the query, based upon the parameters that have been passed to the query at the point the operation is performed. If the original query object had two parameters passed to it, the object that is new or assigned to should have those same two parameters initialized. After either of these operations the two d_OQL_Query objects should be equivalent and have identical behavior.

Example:

Among the math students (as computed before in Section 5.4.1 into the variable mathematicians) who are teaching assistants and earn more than x, find the set of professors that they assist. Suppose there exists a named set of teaching assistants called "TA".

```
d_Bag<d_Ref<Student> >  mathematicians; // computed as above
d_Bag<d_Ref<Professor> >  assisted_profs;
double x = 50000.00;


d_OQL_Query q1 (
      "select t.assists.taught_by from t in TA where t.salary > $1 and t in $2");
q1 << x << mathematicians;
d_oql_execute(q1, assisted_profs);
```

After the above code has been executed, it could be followed by another query, passing in different arguments.

```
d_Set<d_Ref<Student> >  historians;     // assume this gets computed similar
                                        // to mathematicians
double y = 40000.00


q1 << y << historians;
d_oql_execute(q1, assisted_profs);
```

The ODMG OQL implementation may have parsed, compiled, and optimized the original query; it can now reexecute the query with different arguments without incurring the overhead of compiling and optimizing the query.

5.5 Schema Access

This section describes an interface for accessing the schema of an ODMG database via a C++ class library. The schema information is based on the Metadata described in Chapter 2. The C++ schema definition differs in some respects from the language-independent, abstract specification of the schema in ODL. Attempts have been made to make the schema access conform to C++ programming practices in order that the use of the API be intuitive for C++ programmers. C++-specific ODL extensions have been included in the API, in addition to the abstract ODL schema, since the C++ ODL is a superset of the ODMG ODL described in Chapter 2.

The schema access API is structured as an object-oriented framework. Only the interface methods through which meta-information is manipulated is defined, rather than defining the entire class structure and details of the internal implementation. The ODBMS implementation can choose the actual physical representation of the schema database. The ODL schema lists the classes that are used to describe a schema. It also uses ODL relationships to show how instances of these classes are interrelated. To allow maximum flexibility in an implementation of this model, methods are provided to traverse the relationships among instances of these classes, rather than mapping the ODL relationships directly to C++ data members.

The specification currently contains only the *read* interface portion of the schema API. The ODMG plans to extend the specification to include a full *read/write* interface enabling dynamic modification of the schema (e.g., creation and modification of classes). The initial *read-only* interface will substantially increase the flexibility and usability of the standard. The following application domains will be able to take advantage of this interface:

- Database tool development (e.g., class and object browsers, import/export utilities, Query By Example implementations)
- CASE tools
- Schema management tools
- Distributed computing and dynamic binding, object brokers (CORBA)
- Management of extended database properties (e.g. access control and user authorization)

This concept is analogous to the system table approach used by relational database systems. However, since it is a true object mapping, it also includes schema semantics (e.g., relationships between classes and properties), making it much easier to use.

5.5.1 The ODMG Schema Access Class Hierarchy

This section defines the types in the ODMG schema access class hierarchy. Subclasses are indented and placed below their base class. The names in square brackets denote additional base classes, if the class inherits from more than one class. Types in *italic code* font are abstract classes and cannot be directly instantiated; classes in code font can be instantiated. These classes are described in more detail in later sections using both C++ and ODL syntax.

- *d_Scope*
  d_Scope instances are used to form a hierarchy of meta objects. A *d_Scope* instance contains a list of *d_Meta_Object* instances, which are defined in the scope; operations to manage the list (e.g. finding a *d_Meta_Object* by its name) are provided. All meta objects are defined in a scope.

- *d_Meta_Object*
  Instances of *d_Meta_Object* are used to describe elements of the schema stored in the dictionary.
  - d_Module [*d_Scope*]
    d_Module instances manage domains in a dictionary. They are used to group and order *d_Meta_Object* instances, such as type/class descriptions, constant descriptions, subschemas (expressed as d_Module objects), etc.
  - d_Type
    d_Type is an abstract base class for all type descriptions.
    - d_Class [*d_Scope*]
      A d_Class instance is used to describe an application-defined class whose d_Attribute and d_Relationship instances represent the concrete state of an instance of that class. The state is stored in the database. All persistent-capable classes are described by a d_Class instance.
    - d_Ref_Type
      d_Ref_Type instances are used to describe types that are references to other objects. References can be pointers, references (like d_Ref<T>), or other language-specific references. The referenced object or literal can be shared by more than one reference, i.e., multiple references can reference the same object.
    - d_Collection_Type
      A d_Collection_Type describes a type whose instances group a set of elements in a collection. The collection elements must be of the same (base) type.
      - d_Keyed_Collection_Type
        A d_Keyed_Collection_Type describes a collection that can be accessed via keys.
    - d_Primitive_Type
      A d_Primitive_Type represents all built-in types, e.g., int (16, 32 bit), float, etc., as well as pre-defined ODMG literals such as d_String.
      - d_Enumeration_Type [*d_Scope*]
        d_Enumeration_Type describes a type whose domain is a list of identifiers.
    - d_Structure_Type [*d_Scope*]
      d_Structure_Type instances describe application-defined member values. The members are described by d_Attribute instances and

represent the state of the structure. Structures do not have object identity.

- d_Alias_Type
  A d_Alias_Type describes a type that is equivalent to another *d_Type*, but has another name.

- d_Property
  d_Property is an abstract base class for all *d_Meta_Object* instances that describe the state (abstract or concrete) of application-defined types.

  - d_Relationship
    Instances of d_Relationship describe relationships between persistent objects; in C++ these are expressed as d_Rel_Ref<T, MT>, d_Rel_Set<T, MT>, and d_Rel_List<T, MT> data members of a class.

  - d_Attribute
    A d_Attribute instance describes the concrete state of an object or structure.

- d_Operation [*d_Scope*]
  d_Operation instances describe methods, including their return type, identifier, signature, and list of exceptions.

- d_Exception
  d_Exception instances describe exceptions that are raised by operations represented by instances of d_Operation.

- d_Parameter
  A d_Parameter describes a parameter of an operation. A parameter has a name, a type, and a mode (in, out, inout).

- d_Constant
  A d_Constant describes a value that has a name and a type. The value may not be changed.

- d_Inheritance
  d_Inheritance is used to describe the bidirectional relationship between a base class and a subclass, as well as the type of inheritance used.

5.5.2 Schema Access Interface

The following interfaces describe the external C++ interface of an ODMG 2.0 schema repository. The interface is defined in terms of C++ classes with public methods, without exposing or suggesting any particular implementation.

In the following specification, some hints are provided as to how the ODL interface repository described in Chapter 2 has been mapped into the C++ interface.

All objects in the repository are subclasses of *d_Meta_Object* or *d_Scope*.

Interfaces that return a list of objects are expressed using iterator types, traversal of proposed relationships is expressed via methods that return iterators, access to attributes is provided using accessor functions.

An implementation of this interface is not required to use a particular implementation of the iterator type. We follow a design principle used in STL. Classes that have 1-to-n relationships to other classes define only a type to iterate over this relationship. A conformant implementation of the schema repository can implement these types by means of the d_Iterator type (but is not required to).

The iterator protocol must support at least the following methods. In subsequent sections this set of methods for iterator type *IterType* is referred to as a "constant forward iterator" over type T.

```
                    IterType();
                    IterType(const IterType &);
IterType &          operator=(const IterType &);
int                 operator==(const IterType &) const;
int                 operator!=(const IterType &) const;
IterType &          operator++();
IterType            operator++(int);
const  T &          operator*() const;
```

The operator++ advances the iterator to the next element, and operator* retrieves the element. The iterator is guaranteed to always return the elements in the same order. The iterator in class d_Scope that iterates over instances of type d_Meta_Object would be of type d_Scope::meta_object_iterator.

The following names are used for embedded types that provide constant forward iteration.

Iterator Type Name	Element Type
alias_type_iterator	d_Alias_Type
attribute_iterator	d_Attribute
collection_type_iterator	d_Collection_Type
constant_iterator	d_Constant
exception_iterator	d_Exception
inheritance_iterator	d_Inheritance
keyed_collection_type_iterator	d_Keyed_Collection_Type
operation_iterator	d_Operation
parameter_iterator	d_Parameter
property_iterator	d_Property

Iterator Type Name	Element Type
ref_type_iterator	d_Ref_Type
relationship_iterator	d_Relationship
type_iterator	d_Type

Each class that iterates over elements of one of these types has an iterator type defined within the class with the corresponding name. This is denoted in the class interface with a line similar to the following:

```
typedef property_iterator; // this implies an iterator type with this name is defined
```

Properties and classes have access specifiers in C++. Several of the metaclass objects make use of the following enumeration:

```
typedef enum { d_PUBLIC, d_PROTECTED, d_PRIVATE } d_Access_Kind;
```

5.5.2.1 d_Scope

d_Scope instances are used to form a hierarchy of meta objects. A *d_Scope* contains a list of *d_Meta_Object* instances that are defined in the scope, as well as operations to manage the list. The method resolve is used to find a *d_Meta_Object* by name. All instances of *d_Meta_Object*, except d_Module, have exactly one *d_Scope* object. This represents a "defined in" relationship. The type *d_Scope*::meta_object_iterator defines a protocol to traverse this relationship in the other direction.

```
class d_Scope {
public:
    const d_Meta_Object &    resolve(const char *name) const;
    typedef                  meta_object_iterator;
    meta_object_iterator     defines_begin() const;
    meta_object_iterator     defines_end() const;
};
```

5.5.2.2 d_Meta_Object

Class *d_Meta_Object* has a name, an id and a comment attribute. Some instances of *d_Meta_Object* are themselves scopes (instances of *d_Scope*); that is, they define a namespace in which other *d_Meta_Object* instances can be identified (resolved) by name. They form a defines/defined_in relationship with other *d_Meta_Object* instances and are their defining scopes. The scope of a *d_Meta_Object* is obtained by the method defined_in.

```
class d_Meta_Object {
public:
    const char *             name() const;
    const char *             comment() const;
    const d_Scope &          defined_in() const;
};
```

5.5.2.3 d_Module

A d_Module manages domains in a dictionary. They are used to group and order *d_Meta_Object* instances such as type/class descriptions, constant descriptions, subschemas (expressed as d_Module objects), etc. A d_Module is also a *d_Scope* that provides client repository services. A module is the uppermost meta object in a naming hierarchy. The class d_Module provides methods to iterate over the various meta objects that can be defined in a module. It is an entry point for accessing instances of *d_Type*, d_Constant, and d_Operation. The *d_Type* objects returned by type_iterator can be asked for the set of d_Operations, *d_Properties*, etc. that describe operations and properties of the module. It is then possible to navigate further down in the hierarchy. For example, from d_Operation, the set of d_Parameter instances can be reached, and so on.

```
class d_Module : public d_Meta_Object, public d_Scope {
public:
    static const d_Module &    top_level(const d_Database &);

    typedef                    type_iterator;
    type_iterator              defines_types_begin() const;
    type_iterator              defines_types_end() const;

    typedef                    constant_iterator;
    constant_iterator          defines_constant_begin() const;
    constant_iterator          defines_constant_end() const;

    typedef                    operation_iterator;
    operation_iterator         defines_operation_begin() const;
    operation_iterator         defines_operation_end() const;
};
```

5.5.2.4 d_Type

d_Type meta objects are used to represent information about data types. They participate in a number of relationships with the other *d_Meta_Objects*. These relationships allow types to be easily administered within the repository and help to ensure the referential integrity of the repository as a whole.

Class *d_Type* is defined as follows:

```
class d_Type : public d_Meta_Object {
public:
    typedef                    alias_type_iterator;
    alias_type_iterator        used_in_alias_type_begin() const;
    alias_type_iterator        used_in_alias_type_end() const;
```

```
    typedef                         collection_type_iterator;
    collection_type_iterator        used_in_collection_type_begin() const;
    collection_type_iterator        used_in_collection_type_end() const;

    typedef                         keyed_collection_type_iterator;
    keyed_collection_type_iterator  used_in_keyed_collection_type_begin() const;
    keyed_collection_type_iterator  used_in_keyed_collection_type_end() const;

    typedef                         ref_type_iterator;
    ref_type_iterator               used_in_ref_type_begin() const;
    ref_type_iterator               used_in_ref_type_end() const;

    typedef                         property_iterator;
    property_iterator               used_in_property_begin() const;
    property_iterator               used_in_property_end() const;

    typedef                         operation_iterator;
    operation_iterator              used_in_operation_begin() const;
    operation_iterator              used_in_operation_end() const;

    typedef                         exception_iterator;
    exception_iterator              used_in_exception_begin() const;
    exception_iterator              used_in_exception_end() const;

    typedef                         parameter_iterator;
    parameter_iterator              used_in_parameter_begin() const;
    parameter_iterator              used_in_parameter_end() const;

    typedef                         constant_iterator;
    constant_iterator               used_in_constant_begin() const;
    constant_iterator               used_in_constant_end() const;
};
```

5.5.2.5 d_Class

A d_Class object describes an application-defined type whose attributes and relationships form the concrete state of an object of that type. The state is stored in the database. All persistent-capable classes are described by a d_Class instance.

d_Class objects are linked in a multiple inheritance graph by two relationships, inherits and derives. The relationship between two d_Class objects is formed by means of one

connecting d_Inheritance object. A d_Class object also indicates whether the database maintains an extent for the class.

A class defines methods, data and relationship members, constants, and types; that is, the class is their defining scope.

Methods, data and relationship members, constants, and types are modeled by a list of related objects of type d_Operation, d_Attribute, d_Relationship, d_Constant, and *d_Type*. These descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The methods resolve_operation, resolve_attribute, or resolve_constant can be used as shortcuts.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns descriptions for methods, data members, relationship members, constants, and types. Methods, data members, relationship members, constants, and types are also accessible via special iterators. The following functions provide iterators that return their elements in their declaration order:

- base_class_list_begin
- defines_attribute_begin
- defines_operation_begin
- defines_constant_begin
- defines_relationship_begin
- defines_type_begin

The class d_Class is defined as follows:

```
class d_Class : public d_Type, public d_Scope {
public:
    typedef                 inheritance_iterator;
    inheritance_iterator    sub_class_list_begin() const;
    inheritance_iterator    sub_class_list_end() const;
    inheritance_iterator    base_class_list_begin() const;
    inheritance_iterator    base_class_list_end() const;

    d_Boolean               persistent_capable() const; // derived from d_Object?

    // these methods are used to return the characteristics of the class
    typedef                 operation_iterator;
    operation_iterator      defines_operation_begin() const;
    operation_iterator      defines_operation_end() const;
    const d_Operation &     resolve_operation(const char *name) const;

    typedef                 attribute_iterator;
    attribute_iterator      defines_attribute_begin() const;
```

```
    attribute_iterator      defines_attribute_end() const;
    const d_Attribute &     resolve_attribute(const char *name) const;

    typedef                 relationship_iterator;
    relationship_iterator   defines_relationship_begin() const;
    relationship_iterator   defines_relationship_end() const;
    const d_Relationship& resolve_relationship(const char *name) const;

    typedef                 constant_iterator;
    constant_iterator       defines_constant_begin() const;
    constant_iterator       defines_constant_end() const;
    const d_Constant &      resolve_constant(const char *name) const;

    typedef type_iterator;
    type_iterator           defines_type_begin() const;
    type_iterator           defines_type_end() const;
    const d_Type &          resolve_type(const char *name) const;

    d_Boolean               has_extent() const;
};
```

5.5.2.6 d_Ref_Type

d_Ref_Type instances are used to describe types that are references to other types. References can be pointers, references (like d_Ref<T>), or other language-specific references. The referenced object or literal can be shared by more than one reference.

```
class d_Ref_Type : public d_Type {
public:
    typedef enum { REF, POINTER } d_Ref_Kind;
    d_Ref_Kind              ref_kind() const;
    const d_Type &          referenced_type() const;
};
```

5.5.2.7 d_Collection_Type

A d_Collection_Type describes a type that aggregates a variable number of elements of a single type and provides ordering, accessing, and comparison functionality.

```
class d_Collection_Type : public d_Type {
public:
    typedef enum {  LIST,
                    ARRAY,
                    BAG,
```

```
                    SET,
                    DICTIONARY,
                    STL_LIST,
                    STL_SET,
                    STL_MULTISET,
                    STL_VECTOR,
                    STL_MAP,
                    STL_MULTIMAP } d_Kind;

    d_Kind          kind() const;
    const d_Type &  element_type() const;
};
```

5.5.2.8 d_Keyed_Collection_Type

A d_Keyed_Collection_Type describes a collection type whose element can be accessed via keys. Examples are dictionaries and maps.

```
class d_Keyed_Collection_Type : public d_Collection_Type {
public:
    const d_Type &  key_type() const;
    const d_Type &  value_type() const;
};
```

5.5.2.9 d_Primitive_Type

d_Primitive_Type objects represent built-in types. These types are atomic; they are not composed of other types.

```
class d_Primitive_Type : public d_Type {
public:
    typedef enum {
            CHAR,
            SHORT,
            LONG,
            DOUBLE,
            FLOAT,
            USHORT,
            ULONG,
            OCTET,
            BOOLEAN,
            ENUMERATION } d_TypeId;

    d_TypeId        type_id() const;
};
```

5.5.2.10 d_Enumeration_Type

A d_Enumeration_Type describes a type whose domain is a list of identifiers.

An enumeration defines a scope for its identifiers. These identifiers are modeled by a list of related d_Constant objects. The d_Constant objects accessed via the iterator returned by defines_constant_begin are returned in their declaration order.

Constant descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_constant can also be used as a shortcut. The inherited iterator *d_Meta_Object*::meta_object_iterator returns enumeration member descriptions of type d_Constant.

The name of the constant descriptions is equivalent to the domain of the enumeration identifiers. All constants of an enumeration must be of the same discrete type. The enumeration identifiers are associated with values of this discrete type. For instance, an enumeration “days_of_week” has the domain “Monday,” “Tuesday,” “Wednesday,” and so on. The enumeration description refers to a list of seven constant descriptions. The names of these descriptions are named “Monday,” “Tuesday,” “Wednesday,” and so on. All these descriptions reference the same type description, here an object of type d_Primitive_Type with the name “int”. The values of the constants are integers, e.g., 1, 2, 3, and so on up to 7, and can be obtained from the constant description.

```
class d_Enumeration_Type : public d_Primitive_Type, public d_Scope{
public:
    typedef constant_iterator;
    constant_iterator       defines_constant_begin() const;
    constant_iterator       defines_constant_end() const;
    const d_Constant &      resolve_constant(const char *name) const;
};
```

5.5.2.11 d_Structure_Type

d_Structure_Type describes application-defined aggregated values. The members represent the state of the structure. Structures have no identity.

A structure defines a scope for its members. These members are modeled using a list of related d_Attribute objects. The member descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_attribute can be used as a shortcut.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns member descriptions of type d_Attribute. Structure members are also accessible via the iterator returned by defines_attribute_begin, which returns them during iteration in the order they are declared in the structure.

```
class d_Structure_Type : public d_Type, public d_Scope {
public:
    typedef                 attribute_iterator;
    attribute_iterator      defines_attribute_begin() const;
    attribute_iterator      defines_attribute_end() const;
    const d_Attribute &     resolve_attribute(const char *name) const;
};
```

5.5.2.12 d_Alias_Type

A d_Alias_Type describes a type that is equivalent to another type, but has another name. The description of the related type is returned by the method alias_type.

The defining scope of a type alias is either a module or a class; the inherited method *d_Meta_Object*::defined_in returns an object of class d_Class or d_Module.

```
class d_Alias_Type : public d_Type {
public:
    const d_Type &          alias_type() const;
};
```

5.5.2.13 d_Property

d_Property is an abstract base class for d_Attribute and d_Relationship. Properties have a name and a type. The name is returned by the inherited method *d_Meta_Object*::name. The type description can be obtained using the method type_of.

Properties are defined in the scope of exactly one structure or class. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Structure_Type or d_Class, respectively.

```
class d_Property : public d_Meta_Object {
public:
    const d_Type &          type_of() const;
    d_Access_Kind           access_kind() const;
};
```

5.5.2.14 d_Attribute

d_Attribute describes a member of an object or a literal. An attribute has a name and a type. The name is returned by the inherited method *d_Meta_Object*::name. The type description of an attribute can be obtained using the inherited method d_Property::type_of.

Attributes may be read-only, in which case their values cannot be changed. This is described in the meta object by the method is_read_only. If an attribute is a static data member of a class, the method is_static returns d_True.

Attributes are defined in the scope of exactly one class or structure. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Class or d_Structure_Type, respectively.

```
class d_Attribute : public d_Property {
public:
    d_Boolean       is_read_only() const;
    d_Boolean       is_static() const;
    unsigned long   dimension() const;
};
```

5.5.2.15 d_Relationship

d_Relationships model bilateral object references between participating objects. In practice, two relationship meta objects are required to represent each traversal direction of the relationship. Operations are defined implicitly to form and drop the relationship, as well as accessor operations for traversing the relationship. The inherited *d_Type* expresses the cardinality. It may be either a d_Rel_Ref, d_Rel_Set, or d_Rel_List; the method rel_kind returns a d_Rel_Kind enumeration indicating the type.

The defining scope of a relationship is a class. The inherited method *d_Meta_Object*::defined_in returns a d_Class object. The method defined_in_class can be used as a shortcut.

```
class d_Relationship : public d_Property {
public:
    typedef enum { REL_REF, REL_SET, REL_LIST } d_Rel_Kind;
    d_Rel_Kind                  rel_kind() const;
    const d_Relationship &      inverse() const;
    const d_Class &             defined_in_class() const;
};
```

5.5.2.16 d_Operation

d_Operation describes the behavior supported by application objects. Operations have a name, a return type, and a signature (list of parameters), which is modeled by the inherited method *d_Meta_Object*::name, a *d_Type* object returned by result_type, and a list of d_Parameter objects (accessible via an iterator). The d_Parameter objects are returned during iteration in the order that they are declared in the operation. Operations may raise exceptions. The list of possible exceptions is described by a list of d_Exception objects (accessible via an iterator).

Operations may have an access specifier. This is described by the method access_kind inherited from d_Property.

An operation defines a scope for its parameters. They can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_parameter can be used as a shortcut.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns a parameter description of type d_Parameter. Parameters are also accessible via a special parameter_iterator.

The defining scope for an operation is either a class or a module.

The inherited method *d_Meta_Object*::defined_in returns a d_Class object.

```
class d_Operation : public d_Meta_Object, public d_Scope {
public:
    const d_Type &          result_type () const;
    d_Boolean               is_static() const;

    typedef                 parameter_iterator;
    parameter_iterator      defines_parameter_begin() const;
    parameter_iterator      defines_parameter_end() const;
    const d_Parameter &     resolve_parameter(const char *name) const;

    typedef                 exception_iterator;
    exception_iterator      raises_exception_begin() const;
    exception_iterator      raises_exception_end() const;
    d_Access_Kind           access_kind() const;
};
```

If the operation is a static member function of a class, the method is_static returns d_True.

5.5.2.17 d_Exception

Operations may raise exceptions. A d_Exception describes such an exception. An exception has a name, which can be accessed using the inherited method *d_Meta_Object*::name, and a type whose description can be obtained using the method exception_type.

A single exception can be raised in more than one operation. The list of operation descriptions can be accessed via an iterator.

The defining scope of an exception is a module. The inherited method *d_Meta_Object*::defined_in returns a d_Module object. The method defined_in_module can be used as a shortcut.

```
class d_Exception : public d_Meta_Object {
```

```
public:
    const d_Type &          exception_type() const;
    typedef                 operation_iterator;
    operation_iterator      raised_in_operation_begin() const;
    operation_iterator      raised_in_operation_end() const;
    const d_Module &        defined_in_module() const;
};
```

5.5.2.18 d_Parameter

d_Parameter describes a parameter of an operation. Parameters have a name, a type, and a mode (in, out and inout). The name is returned by the inherited method *d_Meta_Object*::name.

The type description can be obtained by the method parameter_type. The mode is returned by the method mode.

Parameters are defined in the scope of exactly one operation, and the inherited method *d_Meta_Object*::defined_in returns an object of class d_Operation. The method defined_in_operation can be used as a shortcut.

```
class d_Parameter : public d_Meta_Object {
public:
    typedef enum { IN, OUT, INOUT } d_Mode;
    d_Mode                  mode() const;
    const d_Type &          parameter_type() const;
    const d_Operation &     defined_in_operation() const;
};
```

5.5.2.19 d_Constant

Constants provide a mechanism for statically associating values with names in the repository. Constants are used by enumerations to form domains. In this case, the name of a d_Constant is used as an identifier for an enumeration value. Its name is returned by the inherited method *d_Meta_Object*::name.

Constants are defined in the scope of exactly one module or class. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Module or d_Class, respectively.

```
class d_Constant : public d_Meta_Object {
public:
    const d_Type &          constant_type() const;
    void *                  constant_value() const;
};
```

5.5.2.20 d_Inheritance

d_Inheritance is used to describe the bidirectional relationship between a base class and a subclass, as well as the type of inheritance used. An object of type d_Inheritance connects two objects of type d_Class.

Depending on the programming language, inheritance relationships can have properties. The schema objects that describe inheritance relationships are augmented with information to reproduce the language-specific extensions.

```
class d_Inheritance {
public:
    const d_Class &     derives_from() const;
    const d_Class &     inherits_to() const;

    d_Access_Kind       access_kind() const;
    d_Boolean           is_virtual() const;
};
```

5.6 Example

This section gives a complete example of a small C++ application. This application manages people records. A Person may be entered into the database. Then special events can be recorded: marriage, the birth of children, moving to a new address.

The application comprises two transactions: the first one populates the database, while the second consults and updates it.

The next section defines the schema of the database, as C++ ODL classes. The C++ program is given in the subsequent section.

5.6.1 Schema Definition

For the explanation of the semantics of this example, see Section 3.2.3. Here is the C++ ODL syntax:

```
// Schema Definition in C++ ODL
class City;             // forward declaration
struct Address {
    d_UShort        number;
    d_String        street;
    d_Ref<City>     city;
                    Address();
                    Address(d_UShort, const char*, const d_Ref<City> &);
};
```

```
extern const char _spouse [ ],  _parents [ ],  _children [ ];

class Person : public d_Object {
public:
// Attributes (all public, for this example)
      d_String                    name;
      Address                     address;
// Relationships
      d_Rel_Ref<Person, _spouse>  spouse;
      d_Rel_List<Person, _parents>  children;
      d_Rel_List<Person, _children> parents;
// Operations
                                  Person(const char * pname);
      void                        birth(const d_Ref<Person> &child); // a child is born
      void                        marriage(const d_Ref<Person> &to_whom);
      d_Ref<d_Set<d_Ref<Person> > >ancestors() const; // returns ancestors
      void                        move(const Address &); // move to a new address
// Extent
static  d_Ref<d_Set<d_Ref<Person> > >     people; // a reference to class extent[1]
static  const char * const                extent_name;
};
class City : public d_Object {
public:
// Attributes
      d_ULong                         city_code;
      d_String                        name;
      d_Ref<d_Set<d_Ref<Person> > >   population;  // the people living in this city
// Operations
                                      City(int, const char*);
// Extension
static   d_Ref<d_Set<d_Ref<City> > >  cities;      // a reference to the class extent
static  const char * const            extent_name;
};
```

1. This (transient) static variable will be initialized at transaction begin time (see the application).

5.6.2 Schema Implementation

We now define the code of the operations declared in the schema. This is written in plain C++. We assume the C++ ODL preprocessor has generated a file, “schema.hxx”, which contains the standard C++ definitions equivalent to the C++ ODL classes.

```
// Classes Implementation in C++
#include "schema.hxx"

const char _spouse [ ] = "spouse";
const char _parents [ ] = "parents";
const char _children [ ] = "children";

// Address structure:

Address::Address(d_UShort pnum, const char* pstreet,
                            const d_Ref<City> &pcity)
  : number(pnumber),
    street(pstreet),
    city(pcity)
{ }

Address::Address()
  : number(0),
    street(0),
    city(0)
{ }
// Person Class:
const char * const Person::extent_name = "people";
Person::Person(const char * pname)
  : name(pname)
{
    people->insert_element(this);  // Put this person in the extension
}
void Person::birth(const d_Ref<Person> &child)
{                                        // Adds a new child to the children list
    children.insert_element_last(child);
    if(spouse)
        spouse->children.insert_element_last(child);
}
void Person::marriage(const d_Ref<Person> &to_whom)
{                                    // Initializes the spouse relationship
```

```
    spouse = with;              // with–>spouse is automatically set to this person
}
d_Ref<d_Set<d_Ref<Person> > >  Person::ancestors()
{                               // Constructs the set of all ancestors of this person
    d_Ref<d_Set<d_Ref<Person> > >  the_ancestors =
                                        new d_Set<d_Ref<Person> >;
    int i;
    for( i = 0; i < 2; i++)
        if( parents[i] ) {
            // The ancestors = parents union ancestors(parents)
            the_ancestors–>insert_element(parents[i]);
            d_Ref<d_Set<d_Ref<Person> > >
                grand_parents= parents[i]–>ancestors();
            the_ancestors–>union_with(*grand_parents);
            grand_parents.delete_object();
        }
    return the_ancestors;
}
void Person::move(const Address &new_address)
{                               // Updates the address attribute of this person
    if(address.city)
        address.city–>population–>remove_element(this);
    new_address.city–>population–>insert_element(this);
    mark_modified();[2]
    address = new_address;
}
// City class:

const char * const City::extent_name = "cities";

City::City(d_ULong code, const char * cname)
  : city_code(code),
    name(cname)
{
    cities–>insert_element(this);       // Put this city into the extension
}
```

2. Do not forget it! Notice that it is necessary only in the case where an attribute of the object is modified. When a relationship is updated, the object is automatically marked modified.

5.6.3 An Application

We now have the whole schema well defined and implemented. We are able to populate the database and play with it. In the following application, the transaction Load builds some objects into the database. Then the transaction Consult reads it, prints some reports from it, and makes updates. Each transaction is implemented inside a C++ function.

The database is opened by the main program, which then starts the transactions.

```
#include <iostream.h>
#include "schema.hxx"

static d_Database dbobj;
static d_Database * database = &dbobj;

void Load()
{                           // Transaction that populates the database
    d_Transaction load;
    load.begin();
    // Create both persons and cities extensions, and name them.

    Person::people = new(database) d_Set<d_Ref<Person> >;
    City::cities = new(database) d_Set<d_Ref<City> >;

    database->set_object_name(Person::people, Person::extent_name);
    database->set_object_name(City::cities, City::extent_name);

    // Construct 3 persistent objects from class Person.

    d_Ref<Person> God, Adam, Eve;

    God   = new(database, "Person") Person("God");
    Adam = new(database, "Person") Person("Adam");
    Eve   = new(database, "Person") Person("Eve");

// Construct an Address structure, Paradise, as (7 Apple Street, Garden),
// and set the address attributes of Adam and Eve.

    Address Paradise(7, "Apple", new(database, "City") City(0, "Garden"));

    Adam->move(Paradise);
    Eve->move(Paradise);
```

```
// Define the family relationships
        God–>birth(Adam);
        Adam–>marriage(Eve);
        Adam–>birth(new(database, "Person") Person("Cain"));
        Adam–>birth(new(database, "Person") Person("Abel"));

        load.commit();      // Commit transaction, putting objects into the database
}

static void print_persons(const d_Collection<d_Ref<Person> >& s)
{                                   // A service function to print a collection of persons
        d_Ref<Person> p;
        d_Iterator<d_Ref<Person> > it = s.create_iterator();
        while( it.next(p) ) {
                cout << "--- " << p–>name << " lives in ";
                if (p–>address.city)
                        cout << p->address.city–>name;
                else
                        cout << "Unknown";
                cout << endl;
        }
}

void Consult()
{                               // Transaction that consults and updates the database
        d_Transaction               consult;
        d_List<d_Ref<Person> >      list;
        d_Bag<d_Ref<Person>>        bag;
        consult.begin();
        // Static references to objects or collections must be recomputed
        // after a commit
        Person::people = database–>lookup_object(Person::extent_name);
        City::cities = database–>lookup_object(City::extent_name);
        // Now begin the transaction
        cout << "All the people ....:" << endl;
        print_persons(*Person::people);
        cout << "All the people sorted by name ....:" << endl;
        d_oql_execute("select p from people order by name", list);
        print_persons(list);
        cout << "People having 2 children and living in Paradise ...:" << endl;
```

```
    d_oql_execute(list, "select p from p in people\
            where   p.address.city.name = \"Garden\"\
                    and count(p.children) = 2", bag);
    print_persons(bag);
    // Adam and Eve are moving ...
    Address Earth(13, "Macadam", new(database, "City") City(1, "St-Croix"));
    d_Ref<Person> Adam;
    d_oql_execute("element(select p from p in people\
            where p.name = \"Adam\")", Adam);
    Adam–>move(Earth);
    Adam–>spouse–>move(Earth);
    cout << "Cain's ancestors ...:" << endl;
    d_Ref<Person> Cain = Adam–>children.retrieve_element_at(0);
    print_persons(*(Cain–>ancestors()));
    consult.commit();
}

main()
 {
    database–>open("family");
    Load();
    Consult();
    database–>close();
}
```

11. APPENDIX C - ODMG CHAPTER 6, SMALLTALK BINDING

Chapter 6

Smalltalk Binding

6.1 Introduction

This chapter defines the Smalltalk binding for the ODMG Object Model, ODL, and OQL. While no Smalltalk language standard exists at this time, ODMG member organizations participate in the X3J20 ANSI Smalltalk standards committee. We expect that as standards are agreed upon by that committee and commercial implementations become available that the ODMG Smalltalk binding will evolve to accommodate them. In the interests of consistency and until an official Smalltalk standard exists, we will map many ODL concepts to class descriptions as specified by Smalltalk80.

6.1.1 Language Design Principles

The ODMG Smalltalk binding is based upon two principles: it should bind to Smalltalk in a natural way that is consistent with the principles of the language, and it should support language interoperability consistent with ODL specification and semantics. We believe that organizations who specify their objects in ODL will insist that the Smalltalk binding honor those specifications. These principles have several implications that are evident in the design of the binding described in the body of this chapter.

1. There is a unified type system that is shared by Smalltalk and the ODBMS. This type system is ODL as mapped into Smalltalk by the Smalltalk binding.
2. The binding respects the Smalltalk syntax, meaning the Smalltalk language will not have to be modified to accommodate this binding. ODL concepts will be represented using normal Smalltalk coding conventions.
3. The binding respects the fact that Smalltalk is dynamically typed. Arbitrary Smalltalk objects may be stored persistently, including ODL-specified objects that will obey the ODL typing semantics.
4. The binding respects the dynamic memory management semantics of Smalltalk. Objects will become persistent when they are referenced by other persistent objects in the database and will be removed when they are no longer reachable in this manner.

6.1.2 Language Binding

The ODMG binding for Smalltalk is based upon the OMG Smalltalk IDL binding.[1] As ODL is a superset of IDL, the IDL binding defines a large part of the mapping required by this document. This chapter provides informal descriptions of the IDL binding topics and more formally defines the Smalltalk binding for the ODL extensions, including relationships, literals, and collections.

The ODMG Smalltalk binding can be automated by an ODL compiler that processes ODL declarations and generates a graph of *meta objects*, which model the schema of the database. These meta objects provide the type information that allows the Smalltalk binding to support the required ODL type semantics. The complete set of such meta objects defines the entire *schema* of the database and would serve much in the same capacity as an OMG Interface Repository. This chapter includes a Smalltalk binding for the Meta Object interfaces defined in Chapter 2.

In such a repository, the meta objects that represent the schema of the database may be programmatically accessed and modified by Smalltalk applications, through their standard interfaces. One such application, a *binding generator*, may be used to generate Smalltalk class and method skeletons from the meta objects. This binding generator would resolve the type/class mapping choices that are inherent in the ODMG Smalltalk binding.

The information in the meta objects is also sufficient to *regenerate* the ODL declarations for the portions of the schema that they represent. The relationships between these components are illustrated in Figure 6-1. A conforming implementation must support the Smalltalk output of this binding process; it need not provide automated tools.

6.1.3 Mapping the ODMG Object Model into Smalltalk

Although Smalltalk provides a powerful data model that is close to the one presented in Chapter 2, it remains necessary to precisely describe how the concepts of the ODMG Object Model map into concrete Smalltalk constructions.

6.1.3.1 Object and Literal

An ODMG object type maps into a Smalltalk class. Since Smalltalk has no distinct notion of literal objects, both ODMG objects and ODMG literals may be implemented by the same Smalltalk classes.

---

1. OMG Document 94-11-8, November 16, 1994.

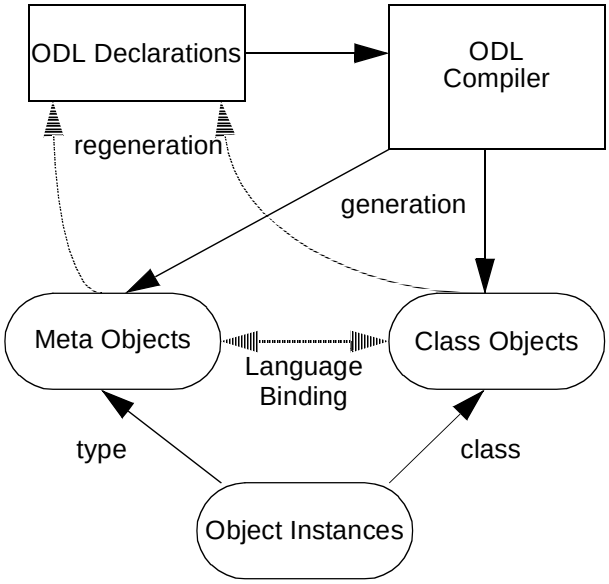


Figure 6-1. Smalltalk Language Binding

6.1.3.2 Relationship

This concept is not directly supported by Smalltalk and must be implemented by Smalltalk methods that support a standard protocol. The relationship itself is typically implemented either as an object reference (one-to-one relation) or as an appropriate Collection subclass (one-to-many, many-to-many relations) embedded as an instance variable of the object. Rules for defining sets of accessor methods are presented that allow all relationships to be managed uniformly.

6.1.3.3 Names

Objects in Smalltalk have a unique object identity, and references to objects may appear in a variety of naming contexts. The Smalltalk system dictionary contains globally accessible objects that are indexed by Symbols which name them. A similar protocol has been defined on the Database class for managing named persistent objects which exist within the database.

6.1.3.4 Extents

Extents are not supported by this binding. Instead, users may use the database naming protocol to explicitly register and access named Collections.

6.1.3.5 Keys

Key declarations are not supported by this binding. Instead, users may use the database naming protocol to explicitly register and access named Dictionaries.

6.1.3.6 Implementation

Everything in Smalltalk is implemented as an object. Objects in Smalltalk have instance variables that are private to the implementations of their methods. An instance variable refers to a single Smalltalk object, the class of which is available at runtime through the class method. This instance object may itself refer to other objects.

6.1.3.7 Collections

Smalltalk provides a rich set of Collection subclasses, including Set, Bag, List, Dictionary, and Array classes. Where possible, this binding has chosen to use existing methods to implement the ODMG Collection interfaces. Unlike statically typed languages, Smalltalk collections may contain heterogeneous elements whose type is only known at runtime. Implementations utilizing these collections must be able to enforce the homogeneous type constraints of ODL.

6.1.3.8 Database Administration

Databases are represented by instances of Database objects in this binding, and a protocol is defined for creating databases and for connecting to them. Some operations regarding database administration are not addressed by this binding and represent opportunities for future work.

6.2 Smalltalk ODL

6.2.1 OMG IDL Binding Overview

Since the Smalltalk/ODL binding is based upon the OMG Smalltalk/IDL binding, we include here some descriptions of the important aspects of the IDL binding that are needed in order to better understand the ODL binding that follows. These descriptions are not intended to be definitions of these aspects, however, and the reader should consult the OMG binding document directly for the actual definitions.

6.2.1.1 Identifiers

IDL allows the use of underscore characters in its identifiers. Since underscore characters are not allowed in all Smalltalk implementations, the Smalltalk/IDL binding provides a conversion algorithm. To convert an IDL identifier with underscores into a Smalltalk identifier, remove the underscore and capitalize the following letter (if it exists):

```
month_of_year
```

in IDL, becomes in Smalltalk:

```
monthOfYear
```

6.2.1.2 Interfaces

Interfaces define sets of operations that an instance that supports that interface must possess. As such, interfaces correspond to Smalltalk protocols. Implementors are free to map interfaces to classes as required to specify the operations that are supported by a Smalltalk object. In the IDL binding, all objects that have an IDL definition must implement a CORBAName method that returns the fully scoped name of an interface that defines all of its IDL behavior.

```
anObject CORBAName
```

6.2.1.3 Objects

Any Smalltalk object that has an associated IDL definition (by its CORBAName method) may be a CORBA object. In addition, many Smalltalk objects may also represent instances of IDL types as defined below.

6.2.1.4 Operations

IDL operations allow zero or more ***in*** parameters and may also return a functional result. Unlike Smalltalk, IDL operations also allow ***out*** and ***inout*** parameters to be defined, which allow more than a single result to be communicated back to the caller of the method. In the Smalltalk/IDL binding, holders for these output parameters are passed explicitly by the caller in the form of objects that support the CORBAParameter protocol (value, value:).

IDL operation signatures also differ in syntax from that of Smalltalk selectors, and the IDL binding specifies a mapping rule for composing default selectors from the operation and parameter names of the IDL definition. To produce the default Smalltalk operation selector, begin with the operation name. If the operation has only one parameter, append a colon. If the operation has more than one parameter, append a colon and then append each of the second to last parameter names, each followed by colon. The binding allows default selectors to be explicitly overridden, allowing flexibility in method naming.

```
current();
days_in_year(in ushort year);
from_hmstz(in ushort hour,
    ushort minute,
    in float second,
    in short tz_hour,
    in short tz_minute);
```

in IDL, become in Smalltalk:

```
current
daysInYear:
fromHmstz:minute:second:tzHour:tzMinute:
```

6.2.1.5 Constants

Constants, Exceptions, and Enums that are defined in IDL are made available to the Smalltalk programmer in a global dictionary CORBAConstants, which is indexed by the fully qualified scoped name of the IDL entity.

```
const Time_Zone USpacific = -8;
```

would be accessed by the Smalltalk:

```
(CORBAConstants at: #'::Time::USpacific)
```

6.2.1.6 Types

Since, in Smalltalk, everything is an object, there is no separation of objects and datatypes as exist in other hybrid languages such as C++. Thus it is necessary for some Smalltalk objects to fill dual roles in the binding. Since some objects in Smalltalk are more natural in this role than others, we will describe the simple type mappings first.

Simple Types

IDL allows several basic datatypes that are similar to literal valued objects in Smalltalk. While exact type-class mappings are not specified in the IDL binding for technical reasons, the following mappings comply:

- short, unsigned short, long, unsigned long — An appropriate Integer subclass (SmallInteger, LargePositiveInteger, LargeNegativeInteger, depending upon the value)
- float, double — Float and Double, respectively
- char — Character
- boolean — The Boolean values **true** and **false**
- octet — SmallInteger
- string — An appropriate String subclass
- any — Object (any class that supports CORBAName)

Compound Types

IDL has a number of data structuring mechanisms that have a less intuitive mapping to Smalltalk. The list below describes the *implicit* bindings for these types. Implementors are also free to provide *explicit* bindings for these types that allow other Smalltalk objects to be used in these roles. These explicit bindings are especially important in the ODL binding since the various Collections have an object-literal duality that is not present in IDL (e.g., ODL list sequences also have a List interface).

- Array — An appropriate Array subclass
- Sequence — An appropriate OrderedCollection subclass
- Structure — Implicit: A Dictionary containing the fields of the structure keyed by the structure fields as Symbols
- Structure — Explicit: A class supporting accessor methods to get and set of the structure fields
- Union — Implicit: Object (any class that supports CORBAName)
- Union — Explicit: A class that supports the CORBAUnion protocol (discriminator, discriminator:, value, and value: methods)
- Enum — A class that supports the CORBAEnum protocol (=, <, > methods). Implementations must ensure that the correct ordering is maintained and that instances of different enumeration types cannot be compared.

Binding Examples

```
union Number switch(boolean) {
   case TRUE:          long integerValue;
   case FALSE:         float realValue;
   };
struct Point{Number x; Number y;};
```

The implicit bindings for the above would allow a Point to be represented by a Dictionary instance containing the keys #x and #y and values that are instances of Integer or Float:

```
aPoint := Dictionary with: #x -> 452 with: #y -> 687.44
```

Alternatively, the binding allows the Smalltalk class Point to represent the struct Point{} because it implements the selectors x, x:, y, and y:.

```
enum Weekday{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};
```

the Smalltalk values for Weekday enumerations would be provided by the implementation and accessed from the CORBAConstants global dictionary, as in:

```
(CORBAConstants at: #'::Date::Wednesday) >
   (CORBAConstants at: #'::Date::Tuesday)
```

6.2.1.7 Exceptions

IDL exceptions are defined within modules and interfaces, and are referenced by the operation signatures that raise them. Each exception may define a set of alternative results that are returned to the caller should the exception be raised by the operation. These values are similar to structures, and Dictionaries are used to represent exception values.

```
exception InvalidDate{};
```

would be raised by the following Smalltalk:

```
(CORBAConstants at: #'::DateFactory::InvalidDate')
   CORBARaiseWith: Dictionary new
```

6.2.2 Smalltalk ODL Binding Extensions

This section describes the binding of ODMG ODL to Smalltalk. ODL provides a description of the database schema as a set of interfaces, including their attributes, relationships, and operations. Smalltalk implementations consist of a set of object classes and their instances. The language binding provides a mapping between these domains.

6.2.2.1 Interfaces and Classes

In ODL, interfaces are used to represent the *abstract behavior* of an object and classes are used to model the *abstract state* of objects. Both types may be implemented by Smalltalk classes. In order to maintain the independence of ODMG ODL and OMG IDL type bindings, all uses of the method CORBAName in the IDL binding are replaced by the method ODLName in this definition. This method will return the name of the ODL interface or class that is bound to the object in the schema repository.

```
aDate ODLName
```

returns the string '::Date', which is the name of its ODL interface. Similarly, all uses of the CORBAConstants dictionary for constants, enums, and exceptions will be replaced by a global dictionary named ODLConstants in this definition. For example:

```
(ODLConstants at: #'::Date::Monday')
```

is a Weekday enum,

```
(ODLConstants at: #'::Time::USpacific')
```

equals -8, and

```
(ODLConstants at: #'::DateFactory::InvalidDate')
```

is an exception.

6.2.2.2 Attribute Declarations

Attribute declarations are used to define pairs of accessor operations that *get* and *set* attribute values. Generally, there will be a one-to-one correspondence between attributes defined within an ODL class and instance variables defined within the corresponding Smalltalk class, although this is not required. ODL attributes define the *abstract state* of their object when they appear within class definitions. When attributes appear within interface definitions, as in IDL they are merely a convenience mechanism for introducing get and set accessing operations.

For example:

```
attribute Enum Rank {full, associate, assistant} rank;
```

yields Smalltalk methods:

```
rank
rank: aProfessorRank
```

6.2.2.3 Relationship Declarations

Relationships define sets of accessor operations for adding and removing associations between objects. As with attributes, relationships are a part of an object's *abstract state*. The Smalltalk binding for relationships results in public methods to *form* and *drop* members from the relationship, plus public methods on the relationship target classes to provide access and private methods to manage the required referential integrity constraints. We begin the relationship binding by applying the Chapter 2 mapping rule from ODL relationships to equivalent IDL constructions, and then illustrate with a complete example.

Single-Valued Relationships

For single-value relationships such as

```
relationship   X   Y              inverse  Z;
```

we expand first to the IDL attribute and operations:

```
attribute      X   Y;
void           form_Y(in X target);
void           drop_Y(in X target);
```

which results in the following Smalltalk selectors:

```
Y
formY:
dropY:
Y:                                "private"
```

For example, from Chapter 3:

```
interface Course {
...
   relationship Professor is_taught_by
        inverse Professor::teaches;
 ...
}
```

yields Smalltalk methods (on the class Course):

```
formIsTaughtBy: aProfessor
dropIsTaughtBy: aProfessor
isTaughtBy
isTaughtBy:                       "private"
```

Multivalued Relationships

For a multivalued ODL relationship such as

```
relationship   set<X>            Y inverse Z;
```

we expand first to the IDL attribute and operations:

```
readonly attribute          set<X>  Y;
void            form_Y(in X target);
void            drop_Y(in X target);
void            add_Y(in X target);
void            remove_Y(in X target);
```

which results in the following Smalltalk selectors:

```
Y
formY:
dropY:
addY:           "private"
removeY:        "private"
```

For example, from Chapter 3:

```
interface Professor {
...
    relationship Set<Course> teaches
        inverse Course::is_taught_by;
...
}
```

yields Smalltalk methods (on class Professor):

```
formTeaches: aCourse
dropTeaches: aCourse
teaches
addTeaches: aCourse         "private"
removeTeaches: aCourse      "private"
```

Finally, to form the above relationship, the programmer could write:

```
| professor course |
professor := Professor new.
course := Course new.
professor formTeaches: course.
    -or-
course formIsTaughtBy: professor.
```

6.2.2.4 Collections

Chapter 2 introduced several new kinds of Collections that extend the IDL sequence to deal with the special needs of ODBMS users. The following shows the Smalltalk method selector that this binding defines for each of the Collection interfaces. Where possible, we have explicitly bound operations to commonly available Smalltalk80 selectors when the default operation binding rules would not produce the desired selector.

“interface Collection”

size	“unsigned long cardinality()”
isEmpty	“boolean is_empty()”
isOrdered	“boolean is_ordered()”
allowsDuplicates	“boolean allows_duplicates()”
add: anObject	“void insert_element(...)”
remove: anObject	“void remove_element(...)”
includes: anObject	“boolean contains_element(...)”
createIterator: aBoolean	“Iterator create_iterator(...)”
createBidirectionalIterator: aBoolean	“BidirectionalIterator create_bidirectional_iterator(...)”

“interface Iterator”

isStable	“boolean is_stable()”
atEnd	“boolean at_end()”
reset	“boolean reset()”
getElement	“any get_element()”
nextPosition	“void next_position()”
replaceElement: anAny	“void replace_element(...)”

“interface BidirectionalIterator”

atBeginning	“boolean at_beginning()”
previousPosition	“void previous_position()”

“interface Set”

createUnion: aSet	“Set create_union(...)”
createIntersection: aSet	“Set create_intersection(...)”
createDifference: aSet	“Set create_difference(...)”
isSubsetOf: aSet	“boolean is_subset_of(...)”
isProperSubsetOf: aSet	“boolean is_proper_subset_of(...)”
isSupersetOf: aSet	“boolean is_superset_of(...)”
isProperSupersetOf: aSet	“boolean is_proper_superset_of(...)”

“interface CollectionFactory”

new: aLong	“Collection new_of_size(...)”

“interface Bag”

occurrencesOf: anAny	“unsigned long occurrences_of(...)”
createUnion: aBag	“Bag create_union(...)”
createIntersection: aBag	“Bag create_intersection(...)”
createDifference: aBag	“Bag create_difference(...)”

“interface List”

at: aULong put: anObject	“void replace_element_at(...)”
removeElementAt: aULong	“void remove_element_at(...)”
retrieveElementAt: aULong	“any retrieve_element_at(...)”
add: anObject after: aULong	“void insert_element_after(...)”
add: anObject before: aULong	“void insert_element_before(.)”
addFirst: anObject	“void insert_element_first(...)”
addLast: anObject	“void insert_element_last(...)”
removeFirst	“void remove_first_element()”
removeLast	“void remove_last_element()”
first	“any retrieve_first_element()”
last	“any retrieve_last_element()”
concat: aList	“List concat(...)”
append: aList	“void append(...)”

“interface Array”

replaceElementAt: aULong element: anAnyObject	“void replace_element_at(...)”
removeElementAt: aULong	“void remove_element_at(...)”
retrieveElementAt: aULong	“any retrieve_element_at(...)”
resize: aULong	“void resize(...)”

“interface Dictionary”

at: anObject put: anObject1	“void bind(...)”
removeKey: anObject	“void unbind(...)”
at: anObject	“any lookup(...)”
includesKey: anObject	“boolean contains_key(...)”

6.2.2.5 Structured Literals

Chapter 2 defined structured literals to represent Date, Time, Timestamp, and Interval values that must be supported by each language binding. The following section defines the binding from each operation to the appropriate Smalltalk selector.

“interface Date”

year	“ushort year()”
month	“ushort month()”
day	“ushort day()”
dayOfYear	“ushort day_of_year()”
monthOfYear	“Month month_of_year()”
dayOfWeek	“Weekday day_of_week()”
isLeapYear	“boolean is_leap_year()”
= aDate	“boolean is_equal(...)”
> aDate	“boolean is_greater(...)”
>= aDate	“boolean is_greater_or_equal(...)”
< aDate	“boolean is_less(...)”
<= aDate	“boolean is_less_or_equal(...)”
isBetween: aDate and: aDate1	“boolean is_between(...)”
next: aWeekday	“Date next(...)”
previous: aWeekday	“Date previous(...)”
addDays: along	“Date add_days(...)”
subtractDays: aLong	“Date subtract_days(...)”
subtractDate: aDate	“Date subtract_date(...)”

“interface DateFactory”

julianDate: aUShort julianDay: aUShort1	“Date julian_date(...)”
calendarDate: aUShort month: aUShort1 day: aUShort2	“Date calendar_date(...)”
isLeapYear: aUShort	“boolean is_leap_year(...)”
isValidDate: aUShort month: aUShort1 day: aUShort2	“boolean is_valid_date(...)”
daysInYear: aUShort	“unsigned short days_in_year(...)”
daysInMonth: aUShort month: aMonth	“unsigned short days_in_month(...)”
today	“Date current()”

“interface Interval”

day	“ushort day()”
hour	“ushort hour()”
minute	“ushort minute()”
second	“ushort second()”
millisecond	“ushort millisecond()”
isZero	“boolean is_zero()”
plus: anInterval	“Interval plus(...)”
minus: anInterval	“Interval minus(...)”
product: aLong	“Interval product(...)”
quotient: aLong	“Interval quotient(...)”
isEqual: anInterval	“boolean is_equal(...)”
isGreater: anInterval	“boolean is_greater(...)”
isGreaterOrEqual: anInterval	“boolean is_greater_or_equal(...)”
isLess: anInterval	“boolean is_less(...)”
isLessOrEqual: anInterval	“boolean is_less_or_equal(...)”

“interface Time”

hour	“ushort hour()”
minute	“ushort minute()”
second	“ushort second()”
millisecond	“ushort millisecond()”
timeZone	“Time_Zone time_zone()”
tzHour	“ushort tz_hour()”
tzMinute	“ushort tz_minute()”
= aTime	“boolean is_equal(...)”
> aTime	“boolean is_greater(...)”
>= aTime	“boolean is_greater_or_equal(...)”
< aTime	“boolean is_less(...)”
<= aTime	“boolean is_less_or_equal(...)”
isBetween: aTime and: aTime1	“boolean is_between(...)”
addInterval: anInterval	“Time add_interval(...)”
subtractInterval: anInterval	“Time subtract_interval(...)”
subtractTime: aTime	“Interval subtract_time(...)”

“interface TimeFactory”

defaultTimeZone	“Time_Zone default_time_zone()”
setDefaultTimeZone	“void setDefault_time_zone(...)”
fromHms: aUShort minute: aUShort1 second: aFloat	“Time from_hms(...)”
fromHmstz: aUShort minute: aUShort1 second: aFloat tzhour: aShort tzminute: aShort1	“Time from_hmstz(...)”
current	“Time current(...)”

“interface Timestamp”

getDate	“Date get_date()”
getTime	“Time get_time()”
year	“ushort year()”
month	“ushort month()”

```
    day                                 "ushort day()"
    hour                                "ushort hour()"
    minute                              "ushort minute()"
    second                              "ushort second()"
    millisecond                         "ushort millisecond()"
    tzHour                              "ushort tz_hour(...)"
    tzMinute                            "ushort tz_minute(...)"
    plus: anInterval                    "Interval plus(...)"
    minus: anInterval                   "Interval minus(...)"
    isEqual: aTimestamp                 "boolean is_equal(...)"
    isGreater: aTimestamp               "boolean is_greater(...)"
    isGreaterOrEqual: aTimestamp        "boolean is_greater_or_equal(...)"
    isLess: aTimestamp                  "boolean is_less(...)"
    isLessOrEqual: aTimestamp           "boolean is_less_or_equal(...)"
    isBetween: aTimestamp
        bStamp: aTimestamp1             "boolean is_between(...)"
"interface TimestampFactory"
    current                             "Timestamp current()"
    create: aDate aTime: aTime          "Timestamp create(...)"
```

6.3 Smalltalk OML

The Smalltalk Object Manipulation Language (OML) consists of a set of method additions to the classes Object and Behavior, plus the classes Database and Transaction. The guiding principle in the design of Smalltalk OML is that the syntax used to create, delete, identify, reference, get/set property values, and invoke operations on a persistent object should be no different from that used for objects of shorter lifetimes. A single expression may thus freely intermix references to persistent and transient objects. All Smalltalk OML operations are invoked by sending messages to appropriate objects.

6.3.1 Object Protocol

Since all Smalltalk objects inherit from class Object, it is natural to implement some of the ODMG language binding mechanisms as methods on this class. The following text defines the Smalltalk binding for the common operations on all objects defined in Chapter 2.

```
"interface Object"
    == anObject                         "boolean same_as(...)"
    copy                                "Object copy()"
    lock: aLockType                     "void lock(...)"
    tryLock: aLockType                  "boolean try_lock(...)"
"interface ObjectFactory"
    new                                 "Object new()"
```

6.3.1.1 Object Persistence

Persistence is not limited to any particular subset of the class hierarchy, nor is it determined at object creation time. A transient object that participates in a relationship with a persistent object will become persistent when a transaction commit occurs. This approach is called *transitive persistence*. Named objects (see “Database Names” on page 216) are the roots from which the Smalltalk binding’s transitive persistence policy is computed.

6.3.1.2 Object Deletion

In the Smalltalk binding, as in Smalltalk, there is no notion of explicit deletion of objects. An object is removed from the database during garbage collection if that object is not referenced by any other persistent object. The delete() operation from interface Object is not supported.

6.3.1.3 Object Locking

Objects activated into memory acquire the default lock for the active concurrency control policy. Optionally, a lock can be explicitly acquired on an object by sending the appropriate locking message to it. Two locking mode enumeration values are required to be supported: read and write. The OMG Concurrency service’s LockSet interface is the source of the following method definitions.

To acquire a lock on an object that will block the process until success, the syntax would be

```
anObject lock: aLockMode.
```

To acquire a lock without blocking, the syntax would be

```
anObject tryLock: aLockMode. "returns a boolean indicating
                              success or failure"
```

In these methods, the receiver is locked in the context of the current transaction. A lockNotGrantedSignal is raised by the lock: method if the requested lock cannot be granted. Locks are released implicitly at the end of the transaction, unless an option to retain locks is used.

6.3.1.4 Object Modification

Modified persistent Smalltalk objects will have their updated values reflected in the ODBMS at transaction commit. Persistent objects to be modified must be sent the message markModified. MarkModified prepares the receiver object by setting a write lock (if it does not already have a write lock) and marking it so that the ODBMS can detect that the object has been modified.

```
anObject markModified
```

It is conventional to send the markModified message as part of each method that sets an instance variable's value. Immutable objects, such as instances of Character and SmallInteger and instances such as **nil**, **true**, and **false**, cannot change their intrinsic values. The markModified message has no effect on these objects. Sending markModified to a transient object is also a null operation.

6.3.2 Database Protocol

An object called a Database is used to manage each connection with a database. A Smalltalk application must open a Database before any objects in that database are accessible. A Database object may only be connected to a single database at a time; however, a vendor may allow many concurrent Databases to be open on different databases simultaneously.

```
"interface Database"
    open: aString                          "void open(...)"
    close                                  "void close()"
    bind: anObject name: aString           "void bind(...)"
    inbind:  aString                       "Object unbind(...)"
    lookup: aString                        "Object lookup(...)"
    schema                                 "Module schema()"
```

6.3.2.1 Opening a Database

To open a new database, send the open: method to an instance of the Database class.

```
database := Database new.
... set additional parameters as required ...
database open: aDatabaseName
```

If the connection is not established, a connectionFailedSignal will be raised.

6.3.2.2 Closing a Database

To close a database, send the close message to the Database.

```
aDatabase close
```

This closes the connection to the particular database. Once the connection is closed, further attempts to access the database will raise a notConnectedSignal. A Database that has been closed may be subsequently reopened using the open method defined above.

6.3.2.3 Database Names

Each Database manages a persistent name space that maps string names to objects or collections of objects, which are contained in the database. The following paragraphs describe the methods that are used to manage this name space. In addition to being assigned an object identifier by the ODBMS, an individual object may be given a name that is meaningful to the programmer or end user. Each database provides methods for

associating names with objects and for determining the names of given objects. Named objects become the roots from which the Smalltalk binding's transitive persistence policy is computed.

The bind:name: method is used to name any persistent object in a database.

```
aDatabase bind: anObject name: aString
```

The lookup:ifAbsent: method is used to retrieve the object that is associated with the given name. If no such object exists in the database, the absentBlock will be evaluated.

```
aDatabase lookup: aString ifAbsent: absentBlock
```

6.3.2.4 Schema Access

The schema of a database may be accessed by sending the schema method to a Database instance. This method returns an instance of a Module that contains (perhaps transitively) all of the meta objects that define the database's schema.

6.3.3 Transaction Protocol

"interface Transaction"	
begin	"void begin()"
commit	"void commit()"
abort	"void abort()"
checkpoint	"void checkpoint()"
isOpen	"boolean isOpen()"
join	"void join()"
leave	"void leave()"
"interface TransactionFactory"	
current	"Transaction current()"

6.3.3.1 Transactions

Transactions are implemented in Smalltalk using methods defined on the class Transaction. Transactions are dynamically scoped and may be started, committed, aborted, checkpointed, joined, and left. The default concurrency policy is pessimistic concurrency control (see Locking, above), but an ODBMS may support additional policies as well. With the pessimistic policy all access, creation, modification, and deletion of persistent objects must be done within a transaction.

A transaction may be started by invoking the method begin on a Transaction instance.

```
aTransaction begin
```

A transaction is committed by sending it the commit message. This causes the transaction to commit, writing the changes to all persistent objects that have been modified within the context of the transaction to the database.

```
aTransaction commit
```

Transient objects are not subject to transaction semantics. Committing a transaction does not remove transient objects from memory, nor does aborting a transaction restore the state of modified transient objects. The method for executing block-scoped transactions (below) provides a mechanism to deal with transient objects.

A transaction may also be checkpointed by sending it the checkpoint message. This is equivalent to performing a commit followed by a begin, except that all locks are retained and the transaction's identity is preserved.

```
aTransaction checkpoint
```

Checkpointing can be useful in order to continue working with the same objects while ensuring that intermediate logical results are written to the database.

A transaction may be aborted by sending it the abort message. This causes the transaction to end, and all changes to persistent objects made within the context of that transaction will be rolled back in the database.

```
aTransaction abort
```

A transaction is open if it has received a begin but not a commit or a abort message. The open status of a particular Transaction may be determined by sending it the isOpen message.

```
aTransaction isOpen
```

A process thread must explicitly create a transaction object or associate itself with an existing transaction object. The join message is used to associate the current process thread with the target Transaction.

```
aTransaction join
```

The leave message is used to drop the association between the current process thread and the target Transaction.

```
aTransaction leave
```

The current message is defined on the Transaction class and is used to determine the transaction associated with the current process thread. The value returned by this method may be **nil** if there is no such association.

```
Transaction current
```

6.3.3.2 Block-Scoped Transactions

A transaction can also be scoped to a Block to allow for greater convenience and integrity. The following method on class Transaction evaluates aBlock within the context of a new transaction. If the transaction commits, the commitBlock will be evaluated after the commit has completed. If the transaction aborts, the abortBlock will be evaluated after the rollback has completed. The abortBlock may be used to undo any side effects of the transaction on transient objects.

```
Transaction perform: aBlock
   onAbort: abortBlock
   onCommit: commitBlock
```

Within the transaction block, the checkpoint message may be used without terminating the transaction.

6.3.3.3 Transaction Exceptions

Several exceptions that may be raised during the execution of a transaction are defined:

- The noTransactionSignal is raised if an attempt is made to access persistent objects outside of a valid transaction context.
- The inactiveSignal is raised if a transactional operation is attempted in the context of a transaction that has already committed or aborted.
- The transactionCommitFailedSignal is raised if a commit operation is unsuccessful.

6.4 Smalltalk OQL

Chapter 4 defined the Object Query Language. This section describe how OQL is mapped to the Smalltalk language. The current Smalltalk OQL binding is a loosely coupled binding modeled after the OMG Object Query Service Specification. A future binding may include one that is more tightly integrated with the Smalltalk language.

6.4.1 Query Class

Instances of the class Query have four attributes: queryResult, queryStatus, queryString, and queryParameters. The queryResult holds the object that was the result of executing the OQL query. The queryStatus holds the status of query execution. The queryString is the OQL query text to be executed. The queryParameters contains variable/value pairs to be bound to the OQL query at execution.

The Query class supports the following methods:

```
create: aQueryString params: aParameterList     "returns a Query"
evaluate: aQueryString params: aParameterList   "returns query result"
complete                                        "returns enum complete"
incomplete                                      "returns enum incomplete"
```

Instances of the Query class support the following methods:

```
prepare: aParameterList     "no result"
execute: aParameterList     "no result"
getResult                   "returns the query result"
getStatus                   "returns a QueryStatus"
```

The execute: and prepare: methods can raise the QueryProcessingError signal if an error in the query is detected. The queryString may include parameters specified by the form $variable, where variable is a valid Smalltalk integer. Parameter lists may be partially specified by Dictionaries and fully specified by Arrays or OrderedCollections.

Example:

Return all persons older than 45 who weigh less than 150. Assume there exists a collection of people called AllPeople.

```
| query result |
query := Query
    create: 'select name from AllPeople where age > $1 and weight < $2'
    params: #(45 150).
query execute: Dictionary new.
[query getStatus = Query complete] whileFalse: [Processor yield].
result := query getResult.
```

To return all persons older than 45 that weigh less than 170, the same Query instance could be reused. This would save the overhead of parsing and optimizing the query again.

```
query execute: (Dictionary with: 2->170).
[query getStatus = Query complete] whileFalse: [Processor yield].
result := query getResult.
```

The following example illustrates the simple, synchronous form of querying an OQL database. This query will return the bag of the names of customers from the same state as aCustomer.

```
Query
    evaluate: 'select c.name from AllCustomers c where c.address.state = $1'
    params: (Array with: aCustomer address state)
```

6.5 Schema Access

Chapter 2 defined Metadata that define the operations, attributes, and relationships between the meta objects in a database schema. The following text defines the Smalltalk binding for these interfaces.

```
"interface MetaObject"
    name                                "attribute name"
    name: aString
    comment                             "attribute comment"
    comment: aString
    formDefinedIn: aDefiningScope       "relationship definedIn"
    dropDefinedIn: aDefiningScope
    definedIn
    definedIn: aDefiningScope
"interface Scope"
    bind: aString value: aMetaObject    "void bind(...)"
    resolve: aString                    "MetaObject resolve(...)"
    unBind: aString                     "MetaObject un_bind(...)"
"interface DefiningScope"
    formDefines: aMetaObject            "relationship defines"
```

```
    dropDefines: aMetaObject
    defines
    addDefines: aMetaObject
    removeDefines: aMetaObject
    createPrimitiveType: aPrimitiveKind
                                    "PrimitiveType create_primitive_type()"
    createCollectionType: aCollectionKind
        maxSize: anOperand
        subType: aType          "Collection create_collection_type(...)"
    createOperand: aString              "Operand create_operand(...)"
    createMember: aString
        memberType: aType               "Member create_member(...)"
    createCase: aString
        caseType: aType
        caseLabels: aCollection         "UnionCase create_case(...)"
    addConstant: aString
        value: anOperand                "Constant add_constant(...)"
    addTypedef: aString alias: aType    "TypeDefinition add_typedef(...)"
    addEnumeration: aString
        elementNames: aCollection       "Enumeration add_enumeration(...)"
    addStructure: aString
        fields: anOrderedCollection     "Structure add_structure(...)"
    addUnion: aString
        switchType: aType
        cases: aCollection              "Union add_union(...)"
    addException: aString
        result: aStructure              "Exception add_exception(...)"
    removeConstant: aConstant           "void remove_constant(...)"
    removeTypedef: aTypeDefinition      "void remove_typedef(...)"
    removeEnumeration: anEnumeration    "void remove_enumeration(...)"
    removeStructure: aStructure         "void remove_structure(...)"
    removeUnion: aUnion                 "void remove_union(...)"
    removeException: anException        "void remove_exception(...)"
"interface Module"
    addModule: aString                  "Module add_module(...)"
    addInterface: aString
        inherits: aCollection           "Interface add_interface(...)"
    removeModule: aModule               "void remove_module(...)"
    removeInterface: anInterface        "void remove_interface(...)"
"interface Operation"
    formSignature: aParameter           "relationship signature"
    dropSignature: aParameter
    signature
    addSignature: aParameter
    removeSignature: aParameter
    formResult: aType                   "relationship result"
    dropResult: aType
    result
    result: aType
```

formExceptions: anException dropExceptions: anException exceptions addExceptions: anException removeExceptions: anException	“relationship exceptions”
“interface Exception”	
formResult: aStructure dropResult: aStructure result result: aStructure	“relationship result”
formOperations: anOperation dropOperations: anOperation operations addOperations: anOperation removeOperations: anOperation	“relationship operations”
“interface Constant”	
formHasValue: anOperand dropHasValue: anOperand hasValue hasValue: anOperand	“relationship hasValue”
formType: aType dropType: aType type type: aType	“relationship type”
formReferencedBy: aConstOperand dropReferencedBy: aConstOperand referencedBy addReferencedBy: aConstOperand removeReferencedBy: aConstOperand	“relationship referencedBy”
formEnumeration: anEnumeration dropEnumeration: anEnumeration enumeration enumeration: anEnumeration	““relationship enumeration”
value	“any value(...)”
“interface Property”	
formType: aType dropType: aType type type: aType	“relationship type”
“interface Attribute”	
isReadOnly isReadOnly: aBoolean	“attribute isReadOnly”
“interface Relationship”	
formTraversal: aRelationship dropTraversal: aRelationship traversal traversal: aRelationship	“relationship traversal”
getCardinality	“Cardinality getCardinality(...)”

“interface TypeDefinition”

formAlias: aType	“relationship alias”
dropAlias: aType	
alias	
alias: aType	

“interface Type”

formCollections: aCollection	“relationship collections”
dropCollections: aCollection	
collections	
addCollections: aCollection	
removeCollections: aCollection	
formSpecifiers: aSpecifier	“relationship specifiers”
dropSpecifiers: aSpecifier	
specifiers	
addSpecifiers: aSpecifier	
removeSpecifiers: aSpecifier	
formUnions: aUnion	“relationship unions”
dropUnions: aUnion	
unions	
addUnions: aUnion	
removeUnions: aUnion	
formOperations: anOperation	“relationship operations”
dropOperations: anOperation	
operations	
addOperations: anOperation	
removeOperations: anOperation	
formProperties: aProperty	“relationship properties”
dropProperties: aProperty	
properties	
addProperties: aProperty	
removeProperties: aProperty	
formConstants: aConstant	“relationship constants”
dropConstants: aConstant	
constants	
addConstants: aConstant	
removeConstants: aConstant	
formTypeDefs: aTypeDefinition	“relationship typeDefs”
dropTypeDefs: aTypeDefinition	
typeDefs	
addTypeDefs: aTypeDefinition	
removeTypeDefs: aTypeDefinition	

“interface PrimitiveType”

kind	“attribute kind”
kind: aPrimitiveKind	

“interface Interface”

formInherits: anInheritance	“relationship inherits”
dropInherits: anInheritance	
inherits	

```
    addInherits: anInheritance
    removeInherits: anInheritance
    formDerives: anInheritance              "relationship derives"
    dropDerives: anInheritance
    derives
    addDerives: anInheritance
    removeDerives: anInheritance
    addAttribute: aString attrType: aType   "Attribute add_attribute(...)"
    addRelationship: aString
        relType: aType
        relTraversal: aRelationship         "Relationship add_relationship(...)"
    addOperation: aString
        opResult: aType
        opParams: anOrderedCollection
        opRaises: anOrderedCollection1      "Operation add_operation(...)"
    removeAttribute: anAttribute            "void remove_attribute(...)"
    removeRelationship: aRelationship       "void remove_relationship(...)"
    removeOperation: anOperation            "void remove_operation(...)"
"interface Inheritance"
    formDerivesFrom: anInterface            "relationship derivesFrom"
    dropDerivesFrom: anInterface
    derivesFrom
    derivesFrom: anInterface
    formInheritsTo: anInterface             "relationship inheritsTo"
    dropInheritsTo: anInterface
    inheritsTo
    inheritsTo: anInterface
"interface Class"
    extents                                 "attribute extents"
    extents: anOrderedCollection
    formExtender: aClass                    "relationship extender"
    dropExtender: aClass
    extender
    extender: aClass
    formExtensions: aClass                  "relationship extensions"
    dropExtensions: aClass
    extensions
    addExtensions: aClass
    removeExtensions: aClass
"interface Collection"
    kind                                    "attribute kind"
    kind: aCollectionKind
    formMaxSize: anOperand                  "relationship maxSize"
    dropMaxSize: anOperand
    maxSize
    maxSize: anOperand
    formSubtype: aType                      "relationship subtype"
    dropSubtype: aType
```

subtype
subtype: aType

isOrdered — “boolean isOrdered(...)”
bound — “unsigned long bound(...)”

“interface Enumeration”

formElements: aConstant — “relationship elements”
dropElements: aConstant
elements
addElements: aConstant
removeElements: aConstant

“interface Structure”

formFields: aMember — “relationship fields”
dropFields: aMember
fields
addFields: aMember
removeFields: aMember

formExceptionResult: anException — “relationship exceptionResult”
dropExceptionResult: anException
exceptionResult
exceptionResult: anException

“interface Union”

formSwitchType: aType — “relationship switchType”
dropSwitchType: aType
switchType
switchType: aType

formCases: aUnionCase — “relationship cases”
dropCases: aUnionCase
cases
addCases: aUnionCase
removeCases: aUnionCase

“interface Specifier”

name — “attribute name”
name: aString

formType: aType — “relationship type”
dropType: aType
type
type: aType

“interface Member”

formStructureType: aStructure — “relationship structure_type”
dropStructureType: aStructure
structureType
structureType: aStructure

“interface UnionCase”

formUnionType: aUnion — “relationship union_type”
dropUnionType: aUnion
unionType
unionType: aUnion

formCaseLabels: anOperand dropCaseLabels: anOperand caseLabels addCaseLabels: anOperand removeCaseLabels: anOperand	“relationship caseLabels”
“interface Parameter”	
parameterMode parameterMode: aDirection	“attribute parameterMode”
formOperation: anOperation dropOperation: anOperation operation operation: anOperation	“relationship operation”
“interface Operand”	
formOperandIn: anExpression dropOperandIn: anExpression operandIn operandIn: anExpression	“relationship OperandIn”
dropValueOf: aConstant valueOf valueOf: aConstant	“relationship valueOf”
formSizeOf: aCollection dropSizeOf: aCollection sizeOf sizeOf: aCollection	“relationship sizeOf”
formCaseIn: aUnionCase dropCaseIn: aUnionCase caseIn caseIn: aUnionCase	“relationship caseIn”
value	“any value(...)”
“interface Literal”	
literalValue literalValue: anAnyObject	“attribute literalValue”
“interface ConstOperand”	
formReferences: aConstant dropReferences: aConstant references references: aConstant	relationship references”
“interface Expression”	
operator operator: aString	“attribute operator”
formHasOperands: anOperand dropHasOperands: anOperand hasOperands addHasOperands: anOperand removeHasOperands: anOperand	“relationship hasOperands”

6.6 Future Directions

Many people believe that keys and extents are an essential ingredient of database query processing. Implicit extents and keys would be preferable to explicit mechanisms involving named Collections, yet there are challenging engineering issues that must be faced to rationalize these capabilities with the notions of transitive persistence and dynamic storage management herein presented.

A uniform set of Database administration operations would facilitate application portability and allow system administration tools to be constructed that could work uniformly across multiple vendors' database products.

This binding has only touched upon the need for interface regeneration mechanisms. Such mechanisms would allow programmers with existing applications utilizing language-specific and even database-specific ODL mechanisms to produce the interface definitions that would insulate them from the differences between these mechanisms.

12. APPENDIX D – ODMG CHAPTER 7, JAVA BINDING

Chapter 7

Java Binding

7.1 Introduction

This chapter defines the binding between the ODMG Object Model (ODL and OML) and the Java programming language as defined by Version 1.1 of the Java™ Language Specification.

7.1.1 Language Design Principles

The ODMG Java binding is based on one fundamental principle: the programmer should perceive the binding as a single language for expressing both database and programming operations, not two separate languages with arbitrary boundaries between them. This principle has several corollaries evident throughout the definition of the Java binding in the body of this chapter:

- There is a single unified type system shared by the Java language and the object database; individual instances of these common types can be persistent or transient.
- The binding respects the Java language syntax, meaning that the Java language will not have to be modified to accommodate this binding.
- The binding respects the automatic storage management semantics of Java. Objects will become persistent when they are referenced by other persistent objects in the database and will be removed when they are no longer reachable in this manner.

Note that the Java binding provides *persistence by reachability*, like the ODMG Smalltalk binding (this has also been called *transitive persistence*). On database commit, all objects reachable from database root objects are stored in the database.

7.1.2 Language Binding

The Java binding provides two ways to declare persistence-capable Java classes:

- Existing Java classes can be made persistence-capable.
- Java class declarations (as well as a database schema) may automatically be generated by a preprocessor for ODMG ODL.

One possible ODMG implementation that supports these capabilities would be a *post-processor* that takes as input the Java .class file (bytecodes) produced by the Java compiler and produces new modified bytecodes that support persistence. Another

implementation would be a *preprocessor* that modifies Java source before it goes to the Java compiler. Another implementation would be a modified Java interpreter.

We want a binding that allows all of these possible implementations. Because Java does not have all hooks we might desire, and the Java binding must use standard Java syntax, it is necessary to distinguish special classes understood by the database system. These classes are called *persistence-capable classes.* They can have both persistent and transient instances. Only instances of these classes can be made persistent. The current version of the standard does not define how a Java class becomes a persistence-capable class.

7.1.3 Use of Java Language Features

7.1.3.1 Namespace

The ODMG Java API will be defined in a vendor-specific package name.

7.1.3.2 Implementation Extensions

Implementations must provide the full function signatures for all the interface methods specified in the chapter, but may also provide variants on these methods with different types or additional parameters.

7.1.4 Mapping the ODMG Object Model into Java

The Java language provides a comprehensive object model comparable to the one presented in Chapter 2. This section describes the mapping between the two models and the extensions provided by the Java binding.

The following features are not yet supported by the Java binding: relationships, extents, keys, and access to the metaschema.

7.1.4.1 Object and Literal

An ODMG object type maps into a Java object type. The ODMG atomic literal types map into their equivalent Java primitive types. There are no structured literal types in the Java binding.

7.1.4.2 Structure

The Object Model definition of a structure maps into a Java class.

7.1.4.3 Implementation

The Java language supports the independent definition of interface from implementation. Interfaces and abstract classes cannot be instantiated and therefore are not persistence-capable.

7.1.4.4 Collection Classes

The collection objects described in Section 2.3.6 specify collection behavior, which may be implemented using many different collection representations such as hash tables, trees, chained lists, etc. The Java binding provides the following interfaces and at least one implementation for each of these collection objects:

```
public interface Set extends Collection { ... }
public interface Bag extends Collection { ... }
public interface List extends Collection { ... }
```

The iterator interface described in Section 2.3.6 is represented by the Java Enumeration interface.

7.1.4.5 Array

Java provides a syntax for creating and accessing a contiguous and indexable sequence of objects, and a separate class, Vector, for extensible sequences. The ODMG Array collection maps into either the primitive array type, the Java Vector class, or the ODMG VArray class, depending on the desired level of capability.

7.1.4.6 Relationship

ODMG relationships are not yet supported by the Java binding.

7.1.4.7 Extents

Extents are not yet supported by the Java binding. The programmer is responsible for defining a collection to serve as an extent and writing methods to maintain it.

7.1.4.8 Keys

Key declarations are not yet supported by the Java binding.

7.1.4.9 Names

Objects may be named using methods of the Database class defined in the Java OML. The root objects of a database are the named objects; root objects and any objects reachable from them are persistent.

7.1.4.10 Exception Handling

When an error condition is detected, an exception is thrown using the standard Java exception mechanism. The following standard exception types are defined; some are thrown from specific ODMG interfaces and are thus subclasses of ODMGException, and others may be thrown in the course of using persistent objects and are thus subclasses of ODMGRuntimeException.

TransactionInProgressException extends ODMGRuntimeException

Thrown when attempting to call a method within a transaction that must be called when no transaction is in progress.

TransactionNotInProgressException extends ODMGRuntimeException

Thrown when attempting to perform outside of a transaction an operation that must be called when there is a transaction in progress.

TransactionAbortedException extends ODMGRuntimeException

Thrown when the database system has asynchronously terminated the user's transaction due to a deadlock, resource failure, etc. In such cases the user's data is reset just as if the user had called Transaction.abort.

DatabaseNotFoundException extends ODMGException

Thrown when attempting to open a database that does not exist.

DatabaseClosedException extends ODMGException

Thrown when attempting to call a method on a database handle that has been closed or when calling a method on a database ID for which the database is not open and was required to be open.

DatabaseIsReadOnlyException extends ODMGRuntimeException

Thrown when attempting to call a method that modifies a database that is open read-only.

ObjectNameNotFoundException extends ODMGException

Thrown when attempting to get a named object whose name is not found.

DatabaseOpenException extends ODMGException

Thrown when attempting to open a database that is already open.

ObjectNameNotUniqueException extends ODMGException

Thrown when attempting to bind a name to an object when the name is already bound to an existing object.

QueryParameterCountInvalidException extends ODMGException

Thrown when the number of bound parameters for a query does not match the number of placeholders.

QueryParameterTypeInvalidException extends ODMGException

Thrown when the type of a parameter for a query is not compatible with the expected parameter type.

LockNotGrantedException extends ODMGRuntimeException

Thrown if a lock could not be granted. (Note that time-outs and deadlock detection are implementation-defined.)

7.2 Java ODL

This section defines the Java Object Definition Language, which provides the description of the database schema as a set of Java classes using Java syntax. Instances of these classes can be manipulated using the Java OML.

7.2.1 Attribute Declarations and Types

Attribute declarations are syntactically identical to field variable declarations in Java and are defined using standard Java syntax and semantics for class definitions.

The following table describes the mapping of the Object Model types to their Java binding equivalents. Note that the primitive types may also be represented by their class equivalents: both forms are persistence-capable and may be used interchangeably.

Object Model Type	Java Type	Literal?
Long	int (primitive), Integer (class)	yes
Short	short (primitive), Short (class)	yes
Unsigned long	long (primitive), Long (class)	yes
Unsigned short	int (primitive), Integer (class)	yes
Float	float (primitive), Float (class)	yes
Double	double (primitive), Double (class)	yes
Boolean	boolean (primitive), Boolean (class)	yes
Octet	byte (primitive), Integer (class)	yes
Char	char (primitive), Character (class)	yes
String	String	yes
Date	java.sql.Date	no
Time	java.sql.Time	no
TimeStamp	java.sql.TimeStamp	no
Set	interface Set	no
Bag	interface Bag	no
List	interface List	no
Array	array type [] or Vector	no
Iterator	Enumeration	no

The binding maps each unsigned integer to the next larger signed type in Java. The need for this arises only where multiple language bindings access the same database. It is vendor-defined whether or not an exception is raised if truncation or sign problems occur during translations. The Java mappings for the object model types Enum and Interval are not yet defined by the standard.

7.2.2 Relationship Traversal Path Declarations

Relationships are not yet supported in the Java binding.

7.2.3 Operation Declarations

Operation declarations in the Java ODL are syntactically identical to method declarations in Java.

7.3 Java OML

The guiding principle in the design of Java Object Manipulation Language (OML) is that the syntax used to create, delete, identify, reference, get/set field values, and invoke methods on a persistent object should be no different from that used for objects of shorter lifetimes. A single expression may thus freely intermix references to persistent and transient objects. All Java OML operations are invoked by method calls on appropriate objects.

7.3.1 Object Creation, Deletion, Modification, and Reference

7.3.1.1 Object Persistence

In the Java binding, persistence is not limited to any particular subset of the class hierarchy, nor is it determined at object creation time. A transient Java object that is referenced by a persistent Java object will automatically become persistent when the transaction is committed. This behavior is called *persistence by reachability.*

Instances of classes that are not persistence-capable classes are never persistent, even if they are referenced by a persistent object. The value of an attribute whose type is not a persistence-capable class is treated by the database system the same way as a transient attribute (see below).

Nevertheless it is possible to declare an attribute to be transient using the keyword transient of the Java language. That means that the value of this attribute is not stored in the database. Furthermore, reachability from a transient attribute will not give persistence to an object.

For example, a class Person with an attribute currentSomething that must not be persistent must be declared as follows:

```
public class Person {
   public String name;
   transient Something currentSomething;
...}
```

When an object of class Person is loaded into memory from the database, the attribute currentSomething is set (by Java) to the default value of its type.

On transaction abort, the value of a transient attribute can be either left unchanged or set to its default value. The behavior is not currently defined by the standard.

Static fields are treated similarly to transient attributes. Reachability from a static field will not give persistence to an object, and on transaction abort the value of a static field can be either left unchanged or set to its default value.

7.3.1.2 Object Deletion

In the Java binding, as in Java, there is no notion of explicit deletion of objects. An object may be automatically removed from the database if that object is neither named nor referenced by any other persistent object.

7.3.1.3 Object Modification

Modified persistent Java objects will have their updated fields reflected in the object database when the transaction in which they were modified is committed.

7.3.1.4 Object Names

A database application generally will begin processing by accessing one or more critical objects and proceeding from there. These objects are *root objects*, because they lead to interconnected webs of other objects. The ability to name an object and retrieve it later by that name facilitates this start-up capability. Names also provide persistence, as noted earlier.

There is a single flat name scope per database; thus all names in a particular database are unique. A name is not explicitly defined as an attribute of an object. The operations for manipulating names are defined in the Database class in Section 7.3.6.

7.3.1.5 Object Locking

We support explicit locking using methods on the Transaction object.

7.3.2 Properties

The Java OML uses standard Java syntax for accessing attributes and relationships, both of which are mapped to field variables.

7.3.3 Operations

Operations are defined in the Java OML as methods in the Java language. Operations on transient and persistent objects behave identically and consistently with the operational context defined by Java. This includes all overloading, dispatching, expression evaluation, method invocation, argument passing and resolution, exception handling, and compile time rules.

7.3.4 Collection Interfaces

A conforming implementation must provide these collection interfaces:

- Collection
- Set
- Bag
- List

An object database may provide any number of instantiable classes to implement representations of the various Collection interfaces. At a minimum, the database must provide three classes SetOfObject, BagOfObject, and ListOfObject, which implement Set, Bag, and List, respectively.

The collection elements are of type Object. Subclasses of Object, such as class Employee, must be converted when used as Collection elements (Java converts them automatically on insertion into a collection, but requires an explicit cast when retrieved).

Chapter 2 defines a number of collections and the semantics of the operations on them. The following sections specify the Java binding interfaces and methods that map on to these collections. Method names have been chosen to match JavaSoft work in progress on collections where possible.

7.3.4.1 Interface Collection

The remove method returns the object removed from the collection, or null if the object was not present in the collection.

```
public interface Collection
{                                                   // Chapter 2 operations
   public int size();                               // unsigned long cardinality()
   public boolean isEmpty();                        // boolean is_empty()
   public void add(Object obj);                     // void insert_element(...)
   public Object remove(Object obj);                // void remove_element(...)
   public boolean contains(Object obj);             // boolean contains_element(...)
   public Enumeration elements();                   // Iterator create_iterator(...)
   public Object selectElement(String predicate);
   public Enumeration select(String predicate);
   public Collection query(String predicate);
   public boolean existsElement(String predicate);
}
```

7.3.4.2 Interface Set

```
public interface Set extends Collection
{                                                   // Chapter 2 operations
   public Set union(Set otherSet);                  // Set create_union (...)
   public Set intersection(Set otherSet);           // Set create_intersection (...)
   public Set difference(Set otherSet);             // Set create_difference(...)
   public boolean subsetOf(Set otherSet);           // boolean is_subset_of(...)
   public boolean properSubsetOf(Set otherSet);     // boolean is_proper_subset_of(...)
   public boolean supersetOf(Set otherSet);         // boolean is_superset_of(...)
   public boolean properSupersetOf(Set otherSet);   // boolean is_proper_superset_of(...)
}
```

7.3.4.3 Interface Bag

The method occurrences returns the number of times an object exists in the Bag, or zero if it is not present in the Bag.

```
public interface Bag extends Collection
{                                                   // Chapter 2 operations
   public Bag union(Bag otherBag);                  // Bag create_union(...)
   public Bag intersection(Bag otherBag             // Bag create_intersection(...)
   public Bag difference(Bag otherBag);             // Bag create_difference(...)
   public int occurrences(Object obj);
}
```

7.3.4.4 Interface List

The beginning List index value is zero, following the Java convention. The method add that is inherited from Collection will insert the object at the end of the List. The remove method returns the object removed from the List, or null if the object was not present in the List.

```
public interface List extends Collection
{                                                   // Chapter 2 operations
   pubic void add(int index, Object obj)            // void insert_element_before(...)
         throws ArrayIndexOutOfBounds;
   public void put(int index, Object obj)           // void replace_element_at(...)
         throws ArrayIndexOutOfBounds;
   public Object remove(int index)                  // void remove_element_at(...)
         throws ArrayIndexOutOfBounds;
   public Object get(int index)                     // any retrieve_element_at(...)
         throws ArrayIndexOutOfBounds;
   public List concat(List other);                  // List concat(...)
   public void append(List other);                  // void append(...)
}
```

The Chapter 2 operations defined on List but not explicitly specified above can be implemented using methods in the Java binding List interface as follows:

Java bindingmethod	Chapter 2 operation
add(index + 1, obj)	insertElementAfter(index, obj)
add(index, obj)	insertElementBefore(index, obj)
add(0, obj)	insertElementFirst(obj)
add(obj)	insertElementLast(obj)
remove(0)	removeFirstElement()
remove(theList.size() - 1)	removeLastElement()
get(0)	retrieveFirstElement()
get(theList.size() - 1)	retrieveLastElement()

7.3.4.5 Interface Array

The Array type defined in Section 2.3.6.4 is implemented by Java arrays, which are single-dimension and fixed-length, or the Java class Vector, instances of which may be resized, or by a class implementing the VArray interface, instances of which may be queried and are otherwise compatible with other collection operations. The remove method returns the object removed from the VArray, or null if the object was not present in the VArray.

```
public void put(int index, Object obj);          // void replace_element_at(...)
        throws ArrayIndexOutOfBounds;
public Object remove(int index)                  // void remove_element_at(...)
        throws ArrayIndexOutOfBounds;
public Object get(int index);                    // any retrieve_element_at(...)
        throws ArrayIndexOutOfBounds;
public void resize(int newSize);                 // void resize(...)
```

7.3.5 Transactions

Transaction semantics are defined in the object model explained in Chapter 2.

Transactions can be *started, committed, aborted,* and *checkpointed.* It is important to note that all access, creation, and modification of persistent objects and their fields must be done within a transaction.

Transactions are implemented in the Java OML as objects of class Transaction, defined as follows:

```
public class Transaction {
   // Creates new transaction object
   //        and associates it with the caller's thread
   Transaction();
```

```
    // Attaches caller's thread to this existing Transaction;
    //      any previous transaction detached from thread
    public void join();

    // Detaches caller's thread from this Transaction,
    //      without attaching another
    public void leave();

    // Returns the current transaction for the caller's thread, or null if none
    public static Transaction current();

    // Starts (opens) a transaction. Nested transactions are not supported.
    public void begin();

    // Returns true if this transaction is open, otherwise false
    public boolean isOpen();

    // Commits and closes a transaction
     public void commit();

    // Aborts and closes a transaction
    public void abort();

    // Commits a transaction but retains locks and reopens transaction
    public void checkpoint();

    // Upgrade the lock on an object
  public void lock(Object obj, int mode);
  public static final int READ, UPGRADE, WRITE;
  }
```

Before performing any database operations, a thread must explicitly create a transaction object or associate (join) itself with an existing transaction object, and that transaction must be open (through a begin call). All subsequent operations by the thread, including reads, writes, and lock acquisitions, are done under the thread's current transaction. A thread may only operate on its current transaction. For example, a TransactionNotInProgressException is thrown if a thread attempts to begin, commit, checkpoint, or abort a transaction prior to joining itself to that transaction.

A DBMS might permit optimistic, pessimistic, or other locking paradigms; ODMG does not specify this.

Transactions must be explicitly created and started; they are not automatically started on database open, upon creation of a Transaction object, or following a transaction commit or abort.

The creation of a new transaction object implicitly associates it with the caller's thread.

The begin function starts a transaction. Calling begin multiple times on the same transaction object, without an intervening commit or abort, causes the exception TransactionInProgressException to be thrown on the second and subsequent calls. Operations executed before a transaction has been opened, or before reopening after a transaction is aborted or committed, have undefined results; these may raise a TransactionNotInProgressException.

There are three ways in which threads can be used with transactions:

1. An application program may have exactly one thread doing database operations, under exactly one transaction. This is the simplest case, and it certainly represents the vast majority of database applications today. Other application instances on separate machines or in separate address spaces may access the same database under separate transactions.
2. There may be multiple threads, each with its own separate transaction. This is useful when writing a service accessed by multiple clients on a network. The database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers *must not* pass objects from one thread to another one that is running under a different transaction; ODMG does not define the results of doing this. However, strings can always be passed between threads, since they are immutable, and scalar data such as integers can be passed around freely.
3. Multiple threads may share one or more transactions. When a transaction is associated with multiple threads simultaneously, all of these threads are affected by data operations or transaction operations (begin, commit, abort). Using multiple threads per transaction is only recommended for sophisticated programming, because concurrency control must be performed by the programmer through Java synchronization or other techniques on top of the DBMS's transaction-based concurrency control.

Calling commit commits to the database all *persistent object modifications* within the transaction and releases any locks held by the transaction. A persistent object modification is an update of any field of an existing persistent object, or an update or creation of a new named object in the database. If a persistent object modification results in a reference from an existing persistent object to a transient object, the transient object is moved to the database, and all references to it updated accordingly. Note that the act

of moving a transient object to the database may create still more persistent references to transient objects, so its referents must be examined and moved as well. This process continues until the database contains no references to transient objects, a condition that is guaranteed as part of transaction commit.

Calling checkpoint commits persistent object modifications made within the transaction since the last checkpoint to the database. The transaction retains all locks it held on those objects at the time the checkpoint was invoked.

Calling abort abandons all persistent object modifications and releases the associated locks.

In the current standard, transient objects are not subject to transaction semantics. Committing a transaction does not remove from memory transient objects created during the transaction, and aborting a transaction does not restore the state of modified transient objects.

Read locks are implicitly obtained on objects as they are accessed. Write locks are implicitly obtained as objects are modified.

Calling lock upgrades the lock on the given object to the given level, if it is not already at or above that level. It throws LockNotGrantedException if it cannot be granted.

Transaction objects are not long-lived (beyond process boundaries) and cannot be stored in a database. This means that transaction objects may not be made persistent, and that the notion of *long transactions* is not defined in this specification.

7.3.6 Database Operations

The predefined type Database represents a database.

```
public class Database {
    // Access modes
    public static final int notOpen = 0;
    public static final int openReadOnly = 1;
    public static final int openReadWrite = 2;
    public static final int openExclusive = 3;

    // Returns the database the name specified.
    //        Opens database if not already open.
    public static Database open(String name, int accessMode)
                    throws ODMGException;
    public void close() throws ODMGException;
```

```
    // Named object binding and lookup
     public void bind(Object object, String name);
     public Object lookup(String name)
                    throws ObjectNameNotFoundException;
     public void unbind(String name)
                    throws ObjectNameNotFoundException;
}
```

The database object, like the transaction object, is transient. Databases cannot be created programmatically using the Java OML defined by this standard. Databases must be opened before starting any transactions that use the database and closed after ending these transactions.

To open a database, use the open method, which takes the name of the database as its argument. This locates the named database and makes the appropriate connection to it. You must open a database before you can access objects in that database. Attempts to open a database when it has already been opened will result in the throwing of the exception DatabaseOpenException. A DatabaseNotFoundException is thrown if the database does not exist. Some implementations may throw additional exceptions that are also derived from ODMGException. Extensions to the open method will enable some object databases to implement default database names and/or implicitly open a default database when a database session is started. Object databases may support opening logical as well as physical databases. Some implementations may also support being connected to multiple databases at the same time.

To close a database, use the close method, which does appropriate clean-up on the named database connection. After you have closed a database, further attempts to access objects in the database will cause the exception DatabaseClosedException to be thrown. Some implementations may throw additional exceptions that are also derived from ODMGException. The behavior at program termination is vendor-defined if databases are not closed or transactions are not committed or aborted. The effect that closing a database has on open transactions is also vendor-defined.

The bind, unbind, and lookup methods allow manipulating names of objects. An object is accessed by name using the lookup member function. The same object may be bound to more than one name. Binding a previously transient object to a name makes that object persistent. The unbind method removes a name and any association to an object and raises an exception if the name does not exist. The lookup method returns null if the specified name is bound to null and generates an exception if the name does not exist.

7.4 Java OQL

The full functionality of the Object Query Language is available through the Java binding. This functionality can be used through query methods on class Collection or through queries using on a stand-alone OQLQuery class.

7.4.1 Collection Query Methods

The Collection interface has a query member function whose signature is:

```
Collection query(String predicate);
```

This function filters the collection using the predicate and returns the result. The predicate is given as a string with the syntax of the where clause of OQL. The predefined variable this is used inside the predicate to denote the current element of the collection to be filtered.

For example, assuming that we have computed a set of students in the variable Students, we can compute the set of students who take math courses as follows:

```
SetOfObject mathematicians;
mathematicians = Students.query(
                    "exists s in this.takes: s.section_of.name = \"math\" ");
```

The selectElement method has the same behavior except that it may only be used when the result of the query contains exactly one element. The select method returns an Enumeration on the result of a query.

7.4.2 The OQLQuery Class

The class OQLQuery allows the programmer to create a query, pass parameters, execute the query, and get the result.

```
class OQLQuery{
   public OQLQuery(){}
   public OQLQuery(String query){ ... } // You can construct or ...
   public create(String query){ ... }      // assign a query.
   public bind(Object parameter){ ... }
   public Object execute() throws ODMGException { ... }
}
```

This is a generic interface. The parameters must be objects, and the result is an Object. This means that you must use objects instead of primitive types (e.g., Integer instead of int) for passing the parameters. Similarly, the returned data, whatever its OQL type, is encapsulated into an object. For instance, when OQL returns an integer, the result is put into an Integer object. When OQL returns a collection whatever its kind (literal or object), the result is always a Java collection of the same kind (for instance, a List).

As usual, a parameter in the query is noted $*i*, where *i* is the rank of the parameter. The parameters are set using the method bind. The *i*th variable is set by the *i*th call to the bind method. If any of the $*i* are not set by a call to bind at the point execute is called, QueryParameterCountInvalidException is thrown. If the argument is of the wrong type, the QueryParameterTypeInvalidException is thrown. After executing a query, the parameter list is reset. Some implementations may throw additional exceptions that are also derived from ODMGException.

Example: Among the students who take math courses (computed in Section 7.4.1) we use OQL to query the teaching assistants (TA) whose salary is greater than $50,000 and who are students in math (thus belonging to the mathematicians collection). The result we are interested in is the professors who are teaching these students. Assume there exists a named set of teaching assistants called TA.

```
Bag mathematicians;
Bag assistedProfs;
Double x;
OQLQuery query;
mathematicians = Students.query(
    "exists s in this.takes: s.sectionOf.name = \"math\" ");
query = new OQLQuery(
    "select t.assists.taughtBy from t in TA where t.salary > $1 and t in $2 ");
x = new Double(50000.0);
query.bind(x); query.bind(mathematicians);
assistedProfs = (Bag) query.execute();
```